



Природно-математички факултет
Универзитет у Крагујевцу

Милош Ивановић

Паралелно програмирање

Скрипта са примерима

базирано на:

*Michael J. Quinn, Parallel Programming in C with MPI and
OpenMP*

Крагујевац, 2016. године

Милош Ивановић
Паралелно програмирање - скрипта са примерима

Рецензенти
Проф. др Александар Пеулић
Проф. др Бобан Стојановић

Издавач
Природно-математички факултет Крагујевац
Радоја Домановића 12
Крагујевац

Штампа
Графички атеље Сквер, Крагујевац

Тираж
300 примерака

ISBN 978-86-6009-042-5

Садржај

1 Увод	11
1.1 Основни појмови	11
1.2 Супер-рачунар	12
1.2.1 Модерни паралелни рачунари	15
1.2.2 Предности паралелног и дистрибуираног рачунарства	15
1.2.3 Модерни научни метод	16
1.2.4 Еволуција супер-рачунара	17
1.3 Стратегије паралелног програмирања	21
1.3.1 Потрага за конкурентностима	21
1.3.2 Приступи програмирању паралелних рачунара	23
2 Архитектура паралелних рачунара	27
2.1 Архитектура мреже	27
2.1.1 Топологије супер-рачунарских мрежа	28
2.1.2 Дводимензиона решетка	29
2.1.3 Бинарно стабло	30
2.1.4 Хиперстабло	31
2.1.5 Топологија лептира	32
2.1.6 Хиперкоцка	34
2.1.7 <i>Shuffle exchange</i> топологија	35
2.2 Векторски рачунари	37
2.3 Мултипроцесори	41
2.3.1 Централизовани мултипроцесор	41
2.3.2 Дистрибуирани мултипроцесор	42
2.4 Мултикомпјутер	43
2.4.1 Асиметрични мултикомпјутер	43
2.4.2 Симетрични мултикомпјутер	44
2.5 <i>Beowulf (Commodity)</i> кластери	44
2.6 Флинова таксономија	45

3 Пројектовање паралелних алгоритама	47
3.1 Модел задатак/канал	47
3.2 Фостерова методологија	48
3.2.1 Партиционисање	48
3.2.2 Комуникација	49
3.2.3 Агломерација	50
3.2.4 Мапирање	51
3.3 Стабло одлучујања за мапирање задатака на процесоре	52
3.4 Пример провођења топлоте	53
3.4.1 Партиционисање и комуникација	55
3.4.2 Агломерација и мапирање	56
3.4.3 Анализа	56
3.5 Максимум низа	57
3.5.1 Партиционисање и комуникација	57
3.5.2 Агломерација и мапирање	61
3.5.3 Анализа	61
3.6 Проблем n тела	62
3.6.1 Партиционисање и комуникација	62
3.6.2 Може ли ефикасније?	64
3.6.3 Анализа	65
3.6.4 Додавање уноса података	66
4 Програмирање разменом порука	71
4.1 Историјат, концепт и основне MPI функције	71
4.1.1 Историјат	71
4.1.2 Модел размене порука	72
4.1.3 Предности модела размене порука	73
4.1.4 Основне MPI функције	73
4.2 Алгоритам одређивања функције истине логичког кола	74
4.2.1 Агломерација и мапирање	75
4.3 Процедура покретања MPI програма	76
4.3.1 Компајлирање	76
4.3.2 Покретање	77
4.3.3 Мерење перформанси	77
5 Ератостеново сито	79
5.1 Секвенцијални алгоритам и паралелизабилност	79
5.1.1 Извори паралелизма	80
5.2 Циклична или блок декомпозиција?	81
5.2.1 Циклична декомпозиција података	81
5.2.2 Блок декомпозиција података	81
5.2.3 Паралелни алгоритам	83

5.2.4	Анализа перформанси	84
5.3	Унапређења	87
5.3.1	Брисање парних бројева	87
5.3.2	Елиминисање емисије	87
5.3.3	Реорганизовање петљи	88
6	Флојдов алгоритам	91
6.1	Секвенцијални алгоритам и извори паралелелизма	91
6.2	Кораци Фостерове методологије	93
6.2.1	Комуникација	93
6.2.2	Агломерација и мапирање	93
6.3	Блокирајуће MPI_Send и MPI_Recv, застој	94
6.3.1	Функција MPI_Send	94
6.3.2	Функција MPI_Recv	95
6.4	Комплексност и време извршења	97
7	Анализа перформанси	101
7.1	Формула за убрзање и ефикасност	101
7.1.1	Амдалов закон	103
7.1.2	Ограничења Амдаловог закона	104
7.1.3	Амдалов ефекат	104
7.2	Густавсон-Барсисов закон	104
7.3	Карп-Флет метрика	106
7.4	Метрика изоэффикасности	108
8	Класификација документа	113
8.1	Опис проблема, пројектовање	113
8.1.1	Партиционисање и комуникација	113
8.1.2	Агломерација и мапирање	113
8.1.3	Концепт господар/слуга	114
8.1.4	MPI_Abort	116
8.1.5	Креирање комуникатора само за раднике	116
8.2	Неблокирајуће MPI функције	118
8.2.1	Функција MPI_Irecv	118
8.2.2	Функција MPI_Wait	119
8.2.3	Функција MPI_Isend	119
8.2.4	Функција MPI_Probe	119
8.2.5	Функција MPI_Get_count	120
8.3	Додатна побољшања	121
8.3.1	Функција MPI_Testsome	121

9 Монте-Карло методе	123
9.1 Примене Монте-Карло методе, паралелизам	123
9.1.1 Пример рачунања броја π	123
9.1.2 Рачунање одређеног интеграла	125
9.1.3 Паралелизам	126
9.2 Секвенцијални генератори случајних бројева	126
9.2.1 Линеарни конгруентни генератор	127
9.2.2 Лаговани Фибоначи генератор	128
9.3 Паралелни генератор случајних бројева	128
9.3.1 Метода господар-слуга	129
9.3.2 Метода жабљег скока	129
9.3.3 Метода поделе на секвенце	129
9.3.4 Метода параметризације	130
9.4 Неуниформне расподеле	130
9.4.1 Инверзна кумулативна функција дистрибуције	130
9.4.2 Случајне тачке унутар кружнице	131
9.4.3 Експоненцијална расподела	135
9.4.4 Бокс-Милерова трансформација	138
9.4.5 Метода одбацања	139
9.5 Студије случаја	142
9.5.1 Траспорт неутрона	142
9.5.2 Температура тачке на дводимензионој плочи	145
9.5.3 Проблем доделе соба	146
Додаци	151
Додатак I Big Data приступ	153
I.1 Apache Spark оквир	154
I.2 Приближно рачунање броја π	156
I.3 Провођење топлоте методом коначних разлика	157
I.4 Класификација докумената	158
Додатак II Елементи MPI-2 стандарда	161
II.1 Стандардизација MPI-а	161
II.2 Портабилни процес покретања	161
II.3 Паралелне улазно/излазне операције	163
II.3.1 Пример непаралелних У/И операција	163
II.3.2 Пример без улазно/излазних MPI операција	165
II.3.3 MPI У/И операције са одвојеним фајловима	166
II.3.4 Паралелне MPI У/И операције са једним фајлом	168
II.3.5 Коришћење појединачних фајл показивача	170
II.3.6 Употреба експлицитних одступања	171

II.3.7	Неконтинуални приступи и колективне У/И операције	172
II.3.8	Приступни низови смештени у фајловима	176
II.3.9	Дељени низови	176
II.3.10	Неблокирајуће У/И операције и дељене колективне У/И опе- рације	178
II.3.11	Дељени фајл показивачи	179
II.4	Даљински приступ меморији	179
II.4.1	Меморијски оквири	179
II.4.2	Перформансе програма са даљинским приступом меморији	180
II.5	Управљање процесима	184
II.5.1	Креирање процеса	184
II.5.2	Пример паралелног копирања	184

Предговор

Скрипта „Паралелно програмирање“ намењена је студентима информатике Природно-математичког факултета у Крагујевцу, и то првенствено за припрему испита из предмета Паралелно програмирање. Овај предмет се по текућој акредитацији слуша као изборни предмет на IV години Основних студија информатике. Резултат је вишегодишњег интересовања аутора за област програмирања супер-рачунара, као и искуства стеченог кроз наставу и полагање самог предмета.

Скрипта обрађује област програмирања паралелних рачунарских система, као што су кластери и супер-рачунари. Иако су рачунари, захваљујући још увек важећем Муром закону све бржи и бржи, понекад један рачунар није довољан да би се решио неки важан проблем науке и технике. Свакодневни пример је прогноза времена, коју данас није могуће замислiti без метеоролошких модела који се „врте“ на паралелним машинама. Да нема супер-рачунара, временску прогнозу за сутра бисмо добили тек након недељу или месец дана, када би нам била потпуно бескорисна. Примера је још много, а ова књига је ту да одговори на питање како се ове паралелне машине програмирају. У техничком смислу, одабрана је најстандартнија могућа батерија алата, програмски језик C и *de-facto* стандард за програмирање путем порука, MPI (*Message Passing Interface*). Неки примери из књиге, посебно у делу који се тиче Монте-Карло метода, користе програмски језик *Octave/Matlab*, док су примери у Додатку А дати у програмском језику *Python*.

Идеја да се напише уџбеник постоји већ неколико година, али је тек сада дошла свој коначни облик. Наиме, на самом почетку, након уласка предмета у званичну акредитацију ОАС Информатике, био је сасвим довољан светски признати уџбеник аутора Мајкла Квина, под називом *Parallel Programming in C with MPI and OpenMP* [1]. Међутим, како је време пролазило, указало се неколико битних недостатака, тј. појавио се јасан простор за побољшање усвајања градива. Као прво, не постоји литература на српском језику чија је примарна тема паралелно програмирање. Друго, неке делове градива је било потребно додатно појаснити и дати референце ван уџбеника [1], посебно ако се има у виду циљни профил студената информатике и њихов ниво познавања математичких и физичких појмова. Треће, неки примери у [1] нису поткрепљени програмским кодом који демонстрира њихову валидност. И коначно, четврто, како је уџбеник [1] објављен пре више од

10 година, неке околности су се значајно промениле, и то како у контексту MPI-а као главне окоснице, тако и кроз појаву сасвим нових софтверских оквира намењених паралелном програмирању који потичу из тзв. *Big Data* арене. Све ове чињенице су довеле аутора до одлуке да напише скрипту која није само превод светски познатог дела, већ пружа и додатне материјале који су потребни савременом ИТ стручњаку који намерава да се бави програмирањем супер-рачунара, као и других дистрибуираних система.

Уџбеника сигурно не би било, да аутор током свих година од када се Паралелно програмирање предаје није имао пуну подршку колега на Институту за математику и информатику ПМФ-а у Крагујевцу. Овом приликом желим да се захвалим колегама на помоћи да паралелно програмирање на Универзитету добије место које заслужује. Посебну захвалност дuguјем Ненаду Стојановићу и Јовану Јанићијевићу који су помогли око обраде текста и слика. Много су ми користиле и писане белешке са мојих предавања колеге Михаила Обреновића, као и мастер рад колеге Ненада Ацковића који се бави различитим аспектима MPI-2 стандарда. Рецензентима и пријатељима проф. др Александру Пеулићу и проф. др Бобану Стојановићу хвала што су, и поред својих огромних обавеза у настави и на пројектима посветили време да преконтROLИШУ рукопис и укажу на недостатке.

Аутор
Крагујевац, 2016. године

Глава 1

Увод

1.1 Основни појмови

Паралелно рачунарство се дефинише као употреба паралелног рачунара у циљу умањења времена потребног за решавање неког проблема. Паралелно и дистрибуирено рачунарство представља брже решавање проблема коришћењем већег броја процесора. Код **паралелног рачунарства у ужем смислу** имамо дељену меморију између више процесора, док код **дистрибуираног рачунарства** сваки процесор поседује своју локалну меморију, тј. свој локални адресни простор. Главни проблеми који се решавају су партиционисање, синхронизација, зависност, као и балансирање оптерећења. **Паралелни рачунар** је рачунарски систем са више процесора који подржава паралелно програмирање. Постоје две категорије паралелних рачунара:

- **Кластер (мултикомпјутер)** - паралелни рачунар састављен од више рачунара повезаних мрежом. Процесори са више различитих рачунара међусобно могу да комуницирају искључиво разменом порука.
- **Централизовани мултипроцесор** (симетрични мултипроцесори, SMP) - интергисани системи у којима сви процесори имају приступ једној главној, глобалној, меморији која подржава комуникацију и синхронизацију између процесора.

Паралелно програмирање је програмирање у језику који дозвољава да се експлицитно зада како ће се разлагати делови израчунавања, тј. како ће бити извршени конкурентно на различиим процесорима.

Највећу мотивацију за развој паралелних рачунара представљају тзв. **Grand Challenge проблеми**. То су фундаментални проблеми науке и инжењерства, који су толико комплексни да њихово решавање путем нумеричких симулација захтева изузетно брзе и сложене рачунаре. Неки од њих су:

- честична физика,
- прогноза времена и климе,
- глобално загревање,
- дизајн материјала,
- суперпроводљивост,
- нано-уређаји,
- космичке технологије,
- моделовање ткива и органа,
- разбијање шифара и обавештајне услуге,
- обрада података прикупљених преко друштвених мрежа,
- фармацеутска индустрија,
- нафтна индустрија...

Са друге стране, садашње стање развоја интегрисаних кола указује да убрзавање микропроцесора једноставним повећањем радне фреквенције више неће бити могуће. Томе доприносе физичка ограничења минијатуризације, трансмисиона граница бакарног проводника итд. Услед ове чињенице, данас је развој микропроцесора пре свега усмерен на увећање броја процесорских језгара, тиме увећавајући значај паралелног програмирања и конкурентне обраде података уопште. Уколико се не деси неки већи продор у индустрији микроелектронике (као напр. продор у квантном рачунарству), паралелизација остаје једини начин да се приступи било каквој масовној обради.

1.2 Супер-рачунар

Супер-рачунар је рачунар чији је рачунарски капацитет за неколико редова величине већи од рачунара опште намене у датом временском тренутку. Одаље директно следи да сама дефиниција супер-рачунара зависи од времена. Неко практично правило налаже да рачунарска машина мора бити барем 100 пута бржа од стандардног персоналног рачунара да би ушла у категорију супер-рачунара. Када се говори о супер-рачунарима, често се срећу и следеће скраћенице:

- **HPC** - *High Performance Computing*,
- **HEC** - *High End Computing* и

- **CI - Cyberinfrastructure.**

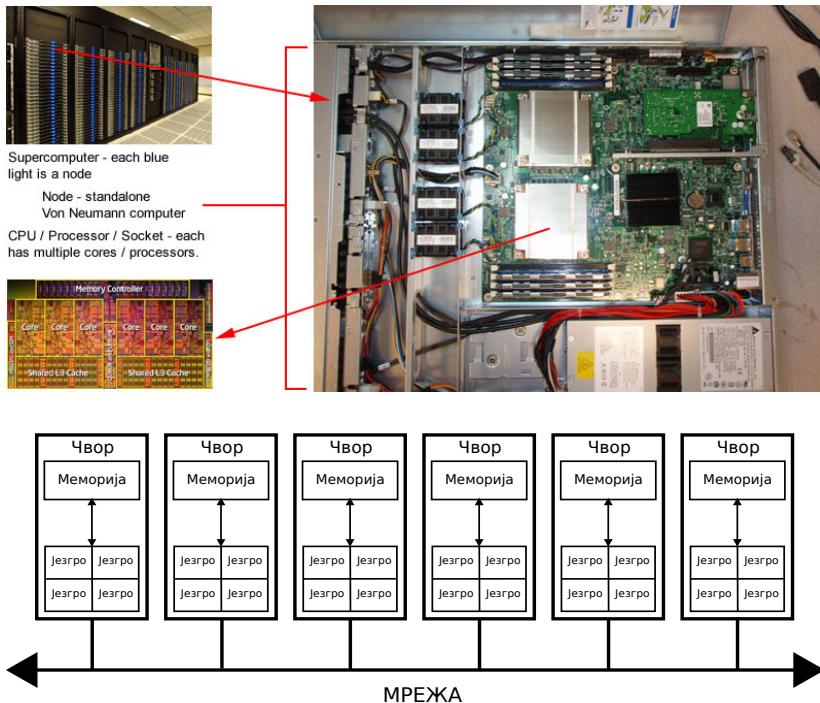
Перформансе супер-рачунара мере се у јединицама под називом FLOPS (*Floating-point Operations per Second*) уместо стандардног MIPS-а (*Million Instructions per Second*). Супер-рачунари који су доступни у току писања овог текста (2016. година) имају капацитет од више десетина PFLOPS (петафлопс). Увек ажурана листа најмоћнијих супер-компјутера данашњице може се наћи на веб локацији www.top500.org.



Слика 1.1: CRAY-1 - пример традиционалног супер-рачунара. Конструисан 1975. године

Традиционални супер-рачунар (Слика 1.1) изграђен је као један једини векторски процесор повезан на меморијску магистралу високих перформанси. Са друге стране, савремени супер-рачунар је паралелни рачунар са хиљадама процесора, спакован у кластер који се састоји од чворова са по више процесора у сваком од њих (Слика 1.2).

Овај историјски прелаз је омогућен развојем микропроцесора и напретком VLSI (*Very Large Scale of Integration*) технологије. За изградњу савремених супер-



Слика 1.2: Архитектура савременог супер-рачунара

рачунара се користи велики број сасвим просечних, на тржишту широко доступних микропроцесора. Традиционални супер-компјутери су били прескупи, док савремени имају далеко бољи однос FLOPS/. Једно илустративно поређење каже да један микропроцесор поседује 1% брзине и 0.1% цене традиционалног супер-рачунара. Одатле следи да је паралелни рачунар од 1000 таквих микропроцесора практично десет пута бржи од традиционалног супер-рачунара, за исту суму новца.

Са друге стране, супер-рачунар ипак није једноставна сума својих процесора. На пример, ако супер-рачунар поседује 1000 процесора, то, на жалост, никако не значи да је 1000 пута бржи. Разлог томе је, логично, спора интерконекција, неадекватно решени улазно/излазни подсистеми, као и неадекватни оперативни системи и програмерска окружења. Све у свему, комерцијални паралелни системи су били прескупи, а услед брзог развоја хардвера су каскали на разним пољима. Са савременог аспекта, синоним за традиционалне супер-рачунаре су примитивна програмерска окружења и искључиви фокус на комерцијални до-мен и владине институције. Научници су, логично, потражили алтернативу.

1.2.1 Модерни паралелни рачунари

Модерни паралелни рачунари су, уместо на специјалним процесорима, засновани на стандардним микропроцесорима који се уградију у персоналне рачунаре и сервере опште намене. У наставку је дато неколико светлих примера са почетка модерног паралелног рачунарства, чији је заједнички именитељ да су засновани на процесорима опште намене:

- **Caltech Cosmic Cube** (1981. година) - састављен од 64 Intel 8086 микропроцесора потпомогнутим 8087 копроцесорима за операције са покретним зарезом. Имао је 128kB меморије по процесору, између 5 и 10 мегафлопса и био дупло јефтинији, а 5-10 пута бржи од тадашњег супер-рачунара који се могао набавити по истој цени.
- **nCube** (1983. година) - комерцијална копија претходног, произвео *Intel*, 512 процесора.
- **Connection Machine (CM2, CM5)** (1987. година) - произвео *Thinking Machines Corporation*, VLSI, 65000 4-битних процесора. CM2-SIMD, CM5-MIMD су подваријанте (Слика 1.3).



Слика 1.3: Thinking Machines Corporation CM-5 (1987. година)

1.2.2 Предности паралелног и дистрибуираног рачунарства

Предности рада на паралелним и дистрибуираним системима могу се сублимирати у следећим ставкама:

1. Решити проблеме брже.
2. Решити обимније проблеме за исто време.
3. Искористити велику количину дистрибуиране меморије.
4. Повећати прецизност решења.
5. Искористи капацитет умрежених радних станица и Грид ресурса.
6. Искористити вишејезгарне процесоре и графичке процесоре.
7. Искористити *dataflow* архитектуре [16].

Наведени бенефити нису поређани према општем приоритету, већ степен зависи искључиво од конкретног проблема. У овој листи посебно треба обратити пажњу на ставке 2 и 3, јер понекад сирова брзина није примарна предност. На пример, ако имамо 10 сати на располагању за решавање неког проблема, понекад и не значи пуно што ће се исти прорачун на супер-рачунару завршити за 10 минута. Међутим, итекако значи чињеница што сада у тих 10 сати, који су иначе на располагању, може да се уради 60 пута више послла и, примера ради, добију далеко прецизнији резултати.

Друго, неке проблеме и није могуће сместити у меморију стандардног рачунара, па се ту прича са њима завршава уколико се проблем не портује на суперрачунар, где владају потпуно другачији редови величине.

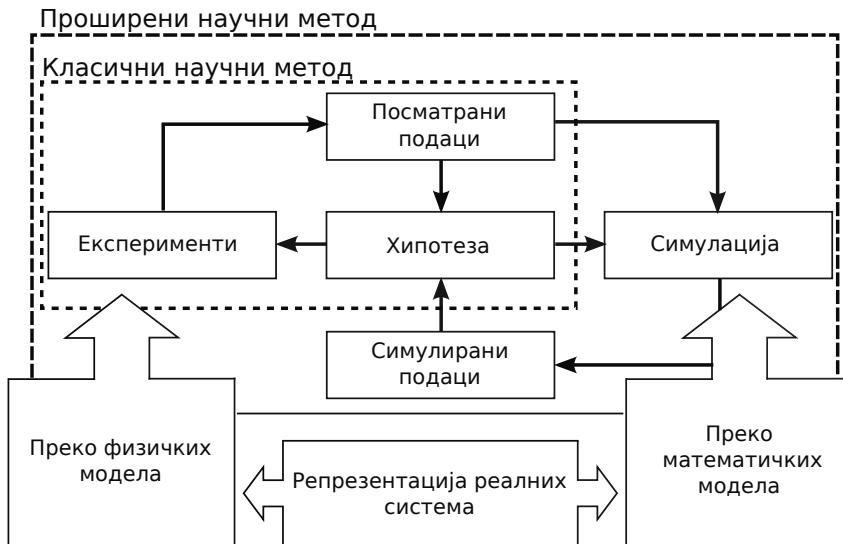
1.2.3 Модерни научни метод

Класични научни приступ - Научник посматра феномен, развија теорију да би га објаснио и изводи експеримент да би истестирао теорију. Експеримент често доводи до промене, или чак потпуног одбацивања теорије, па се научник враћа на посматрање. Класична наука се заснива на физичким експериментима и моделима.

Савремени научни приступ - Заснива се на нумеричким симулацијама (Слика 1.4). Оне замењују физичке експерименте када су прескупи, предуги, неетички или их је просто немогуће извести. Научник пореди резултате нумеричке симулације, засноване на теорији, са подацима прикупљеним из природе.

Уколико потражимо конкретне примере истраживања код којих је експерименте тешко или немогуће изводити, па рачунарске симулације играју доминантну улогу, видећемо да их можемо поделити у неколико категорија:

- **Превелико** - Космологија, формација галаксија, еволуција звезда, прорачун путања космичких сонди...



Слика 1.4: Класични научни метод насупрот модерног научног метода

- **Премало** - Честична физика, секвенцирање гена, моделирање геометрије молекула...
- **Пребрзо** - Фотосинтеза, синтеза протеина...
- **Преспоро** - Геологија, климатске промене...
- **Прекомплексно** - Проток крви у капиарима, моделирање мишића, временска прогноза...
- **Преопасно** - Токсичне супстанце, одржавање нуклеарног арсенала, детекција олуја које могу да произведу торанада...

1.2.4 Еволуција супер-рачунара

Крајем Другог светског рата у Сједињеним Америчким Државама је констуиран ENIAC (*Electronic Numerical Integrator And Computer*), како би се брже и тачније срачунавале тзв. артиљеријске табеле, тј. пројектоване путање пројектила високог дometа. Током Хладног рата, јаки рачунари су коришћени за пројектовање нуклеарног наоружања, разбијање шифара, космички програм и сличне намене. Данас се углавном не развија ново нуклеарно наоружање, већ се користе нумеричке симулације да би се одржавао постојећи арсенал нуклеарног наоружања.

Данас супер-рачунари имају далеко ширу примену, што говори у прилог чињеници да је велика потражња покретач за развој оваквих машина.

MFLOPS	GFLOPS	TFLOPS	PFLOPS	EFLOPS
$10^6, 2^{20}$	$10^9, 2^{30}$	$10^{12}, 2^{40}$	$10^{15}, 2^{50}$	$10^{18}, 2^{60}$

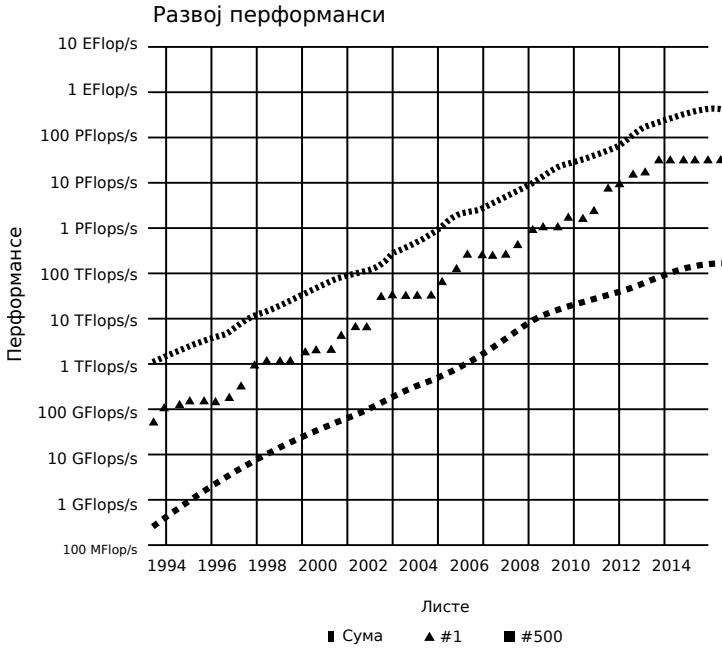


Слика 1.5: Развој супер-рачунара

Као што се види на Сликама 1.5 и 1.6, рачунари су за последњих 60 година убрзани више од трилион пута. Процесори су милионима пута бржи, инструкције се извршавају конкурентно, комбинује се хиљаде процесора. Међутим, треба имати у виду да се не развијају све рачунарске компоненте подједнаком брзином и да збирна брзина рачунара зависи пре свега од избалансираности његових саставних компоненти. На Слици 1.7 може се, опет на логаритамској скали, видети еволуција рачунарских компоненти разврстаних по типу. Јасно је да је компонента која највише заостаје заправо софтвер, услед реалне немогућности да се софтверска индустрија одмах конзумира све предности које индустрија микролектронике доноси.

Традиционални супер-рачунари су коштали преко 10 милиона долара и могли су се наћи искључиво у државним лабораторијама, индустријама са великим капиталом (нафтна, аутомобилска) и осталим јаким гранама привреде (фармацеутска индустрија, финансијске трансакције). Као што је већ напоменуто, научници су потражили алтернативу, како би већој заинтересованој популацији приближили снагу супер-рачунара.

Пошто су се и „обични“ рачунари изузетно брзо развијали, почетком 90-их година прошлог века, NASA-ини научници су искористили широко доступне процесоре, уобичајену интерконекцију и бесплатан софтвер и склопили први тзв. **Beowulf кластер**. За релативно мало улагање су добили веома високе перформансе. Њихов систем се састојао од 16 стандардних рачунара са Intel 486DX4 процесорима који су повезани уобичајеном *Ethernet* мрежом од $10Mbps$. На чврзовима овог кластера био је инсталiran **Linux** оперативни систем, са батеријом GNU компајлера и библиотеком која подлеже MPI стандарду за интерпроцесну комуникацију. На оваквом систему је брзина мреже далеко мања од брзине израчунавања (за разлику од комерцијалних супер-рачунара), али за широку категорију апликација где преовлађују израчунавања, *Beowulf* кластер постиже зна-



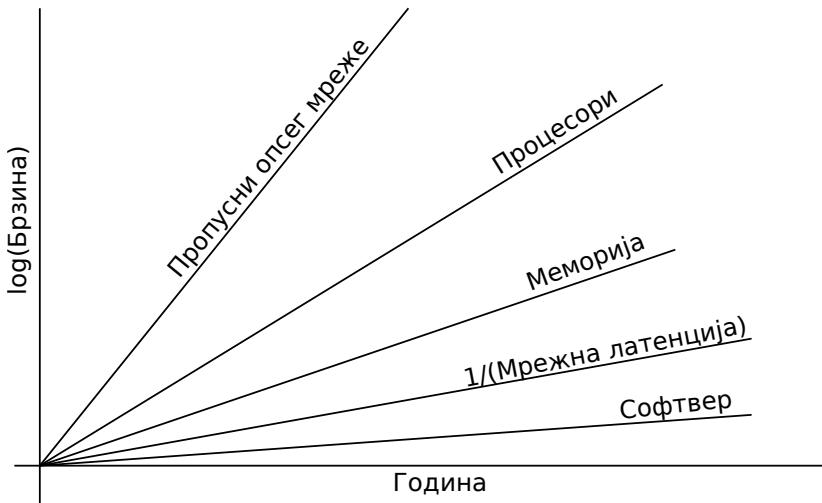
Слика 1.6: Еволуција супер-рачунара на логаритамској скали. Тачкама је означена suma свих супер-рачунара са Top500 листе, троугловима први на тој листи, а квадратима последњи.

чајно бољи однос перформансе/цена од традиционалних супер-рачунара.

Ексафлоп супер-рачунар

Неколико држава (САД, Кина, Индија) је као један од примарних циљева развоја у неколико наредних година поставило развој супер-рачунара који би достигао снагу ексафлопа. И поред великих тешкоћа, као што су дисипација топлоте, ограничење минијатуризације, трансмисиона граница бакарног проводника, ефикасност, итд, врло је вероватно да ће се граница ексафлопа досегнути неким новим технолошким пробојем. Према Густавсону [3], ево шта би се све могло очекивати када рачунари овог реда снаге постану свакодневица:

- Фантастични материјали** - Често се у популарној штампи помињу постигнућа на пољима нових материјала, као што је однос чврстоће и масе који омогућава конструкцију космичког подизача или неки други инжењерски дизајн... Помињу се и напори експерименталних физичара, који у лабораторији постижу суперпроводљивост на рекордно високој температури. Или материјал који омогућава рекордну ефикасност соларних панела... Открића оваквих материјала су веома захтевна са становишта уложеног ла-



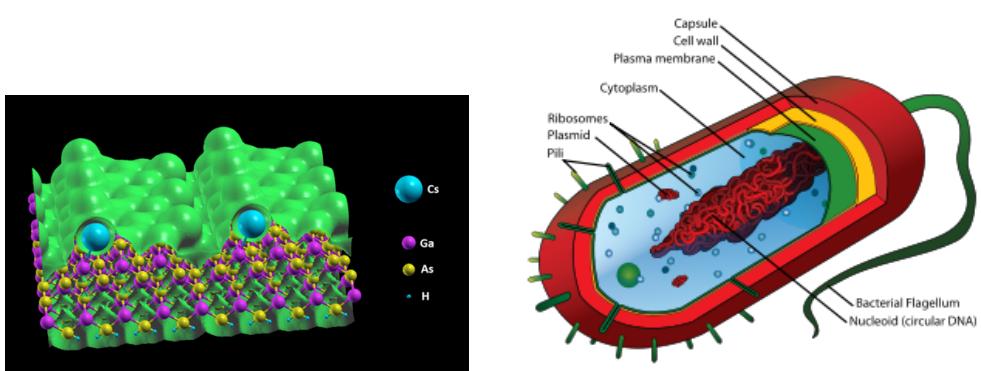
Слика 1.7: Еволуција на рачунарски компоненти

бораторијског рада традиционалном методом пробе и грешке. Међутим, помоћу доволно јаког супер-рачунара, било би могуће испробати понашање ових материјала унутар симулације, која би могла да опонаша атоме и молекуле на квантно-механичком нивоу!

- **Симулација живе ћелије** - Шта бисмо добили моделирањем живота на Земљи, тј. ако бисмо могли да симулирамо сваку активност која се одвија у једној живој ћелији? Ту спадају метаболичке стазе, транспорт материјала, функције протеина, начин репродукције, све то у најситнијим детаљима. Научници ће највероватније прво циљати на најједноставнију ћелију, ћелију прокариоте. Помоћу супер-рачунара који може да одржи неколико ексафлопа, било би могуће симулирати целу ћелију уместо појединачних фрагмената, како је то до сада био случај.
- **Ултимативни дизајн возила** - Пуно је параметара који утичу на квалитет аутомобила и авиона - облик, брзина, отпор ваздуха, бука, безбедност, енергетска ефикасност,... Помоћу ексафлоп супер-рачунара би сви ови параметри могли да се оптимизују **одједном**, уместо да се чекају инжењери и њихова инкрементална унапређења која трају месецима, па чак и годинама.
- **Персонализовано лечење карцинома** - Приступ који већ данас пуно обава је лечење карцинома персонализованом имунотерапијом. Велики број врста карцинома подразумева тзв. мрачни ДНК. Када се он активира, производе се лоши протеини, а наш имуни систем можда неће успети да се

носи са њима. Ако бисмо могли да користимо ексафлоп супер-рачунар да прорачуна све потенцијалне лоше протеине, који су за свакога различити, били бисмо у могућности да направимо персонализовану вакцину!

- Графика филмског квалитета у реалном времену** - Ексаскаларне апликације не морају у сваком случају да имају озбиљну употребу. Ако узмемо за пример колико је компјутерске снаге требало да се уради анимирани филм *Monsters University*, једноставном рачуницом долазимо до бројке од 6 ексафлопа, уколико бисмо желели да се фрејмови рачунају у реалном времену (док гледамо филм). То значи да би са таквом снагом било могуће добити виртуелну реалност према којој данашње компјутерске игре изглеђају као стари Пекмен. Другим речима, помоћу ексафлоп супер-рачунар би било могуће у потпуности преварити наша чула (барем чуло вида и чуло слуша) и иззбрисати границу између реалног и виртуелног света.



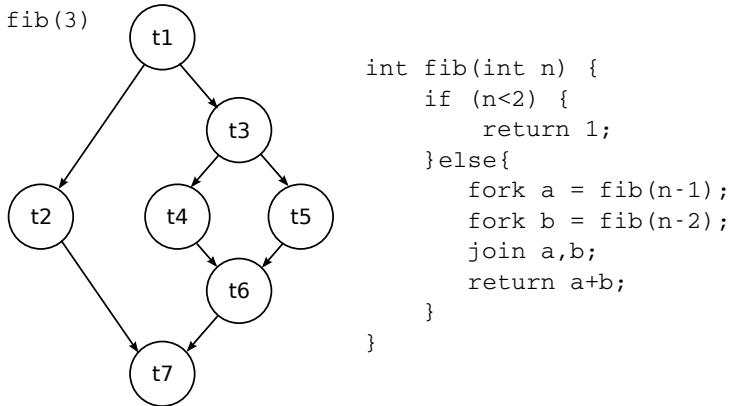
Слика 1.8: Лево: Структура материјала. Десно: Шематски приказ ћелије прокариоте

1.3 Стратегије паралелног програмирања

1.3.1 Потрага за конкурентностима

Граф зависности је усмерени граф у којем сваки чвр представља један задатак, док везе представљају зависности између њих. Стрелица из чвора U у чвор V значи да задатак U мора бити извршен пре него што задатак V почне. Ако нема путање из U у V , значи да су задаци независни и могу да се изврше истовремено. На Слици 1.9 дат је пример графа зависности који се користи приликом

рекурзивног рачунања Фибоначијевог низа¹.



Слика 1.9: Граф зависности код рачунања Фибоначијевог низа

Паралелизам података је тип паралелизма у коме независни процеси примењују исту операцију на различите елементе скупа података. Сви таскови могу да се извршавају конкурентно. Уколико је са p означен број процесора, брзина може да се повећа и до p пута. На пример:

```
for (i=0; i<100; ++i)
    a[i]=b[i]+c[i];
```

Јасно је да свих 100 итерација наведене петље могу да се извршавају истовремено.

Функционални паралелизам је тип паралелизма у коме независни подзадаци извршавају функције над истим или различитим елементима података. Степен конкурентности је лимитиран бројем конкурентних подзадатака. На пример:

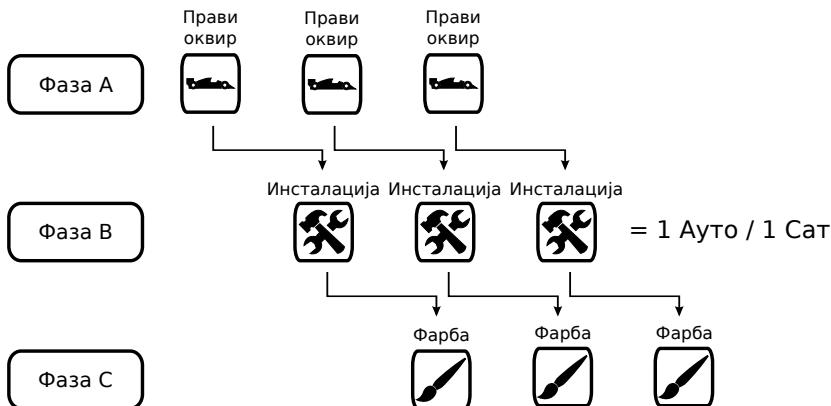
```
a = 2;
b = 3;
m = (a+b)/2;
s = (a*2+b*2)/2;
v = s-m*2;
```

Овде трећи и четврти израз могу да се изврше паралелно.

Код приступа **цевовода** (*pipeline*) нема ни класичног паралелизма података, а ни функционалног паралелизма. У случају да се решава само једна инстанца проблема, не можемо да добијемо никакав паралелизам. Међутим, ако је потребно решити више инстанци истог проблема, читав проблем се дели на фазе ($A, B,$

¹Фибоначијев низ се дефинише преко $F_n = F_{n-1} + F_{n-2}$, где су $F_1 = F_2 = 1$

C, ...), где је сваки процесор задужен за по једну фазу. Ово доводи до резултата да се обрађује неколико објеката истовремено, као на траци за склапање аутомобила (Слика 1.10). Убрзање је лимитирано бројем фаза цевовода.



Слика 1.10: Паралелизам цевовода објашњен на примеру производње аутомобила

1.3.2 Приступи програмирању паралелних рачунара

У савременом паралелном рачунарству, присутно је, у мањој или већој мери, четири практична приступа, и то:

1. **Проширење компајлера** у смислу додавања могућности да секвенцијалне програме аутоматски преводи у паралелне.
2. **Проширење постојећег језика**, додавање нових паралелних операција.
3. **Додавање новог паралелног слоја** на постојећи секвенцијални језик.
4. **Увођење потпуно новог језика** који природно подржава паралелизам.

У наставку ће бити размотрене предности и мање сваког од наведених приступа.

Проширење компајлера

Задатак је модификација постојећег компајлера додавањем могућности да аутоматски детектује паралелизам у секвенцијалним програмима и да одатле произведе паралелни извршни фајл. На овом пољу је у прошлости било веома много истраживања, а фокус је стављен на компјалере за програмски језик *Fortran*.

Предности оваквог приступа су у томе се може се применити на постојећи код, аутоматска паралелизација би уштедела време и напор. Затим, програмери не би морали да паралелизују свој код, нити да уче паралелно програмирање, већ само да наставе да користе једноставније секвенцијалне језике, а да паралелизацију оставе компајлеру и оперативном систему.

Мане овог приступа су што, пре свега, паралелизам може да се неповратно изгуби када се програм напише секвенцијално. Уколико је програмер закомпликовао код, компајлер са великом вероватноћом неће моћи ни да пронађе потенцијални паралелизам. Дакле, овај приступ функционише искључиво код једноставних конструкција, петљи исл. Развијени су многи експериментални компајлери овог типа, али ниједно решење продукционог квалитета.

Проширење језика

Овај приступ се своди на додавање паралелних функција у постојећи секвенцијални језик. То су функције за креирање и терминирање паралелних процеса, њихову синхронизацију, међусобну комуницирају исл. Примери из праксе су MPI, PVM, POSIX нити, OpenMP идр.

Овај приступ је најлакши, најбржи, најефтинији и најпопуларнији приступ паралелним програмирању. Захтева само развој библиотеке рутине. Могуће је користити већ постојеће језике и компајлере, док се нове паралелне библиотеке развијају релативно брзо и једноставно. На пример, MPI библиотеке постоје за готово сваки паралелни рачунар.

Мана оваквог приступа је, пре свега, у томе што компајлери не учествују у генерисању паралелног кода, нити омогућавају хватање грешака. Због тога је прилично тешко отклањати грешке, чак и у једноставним програмима.

Додавање паралелног програмског слоја

Паралелни програм можемо да посматрамо као да је двослојан. Доњи слој је компјутационо језгро, у коме процес манипулише сопственом порцијом података да би произвео своју порцију резултата. Овај слој може да се имплементира у постојећем секвенцијалном програмском језику. Горњи слој контролише креирање и синхронизацију процеса, као и партиционисање података међу процесима. Паралелни слој може да буде испрограмиран неким паралелним језиком. Развијено је неколико истраживачких прототипова, али ниједан комерцијални систем овог типа није заживео.

Паралелни програмски језик

Један начин је развијање паралелног програмског језика. Пример је језик *Octave*, са потпуно новом синтаксом, који подржава и паралелно и секвенцијално извршавање процеса. Други начин је додавање паралелних конструкција у већ

постојећи програмски језик. Примери су *High Performance Fortran* и *C**. Ту је и компајлерско решење CUDA компаније *nVidia*, које додаје специјалне инструкције за програмирање масивно паралелних графичких процесора.

Предност овог приступа је што програмер предочава паралелизам самом компајлеру, што повећава вероватноћу да ће извршни програм достићи високе перформансе.

Основна мана је да се захтева развој нових компајлера. Произвођачима су потребне године за развој квалитетног компајлера за свој паралелни систем. Друго, нови језици можда и неће бити стандардизовани (пример из прошлости је *C**) и онда произвођачи одлучују да не праве компајлер за те језике на својим машинама. Последње, али не и најмање важно, програмери имају подразумевани отпор према учењу нових конструкција језика.

Актуелно стање

Док се рад на развијању паралелизујућих компајлера и паралелних програмских језика високог нивоа наставља, најпопуларнији приступ остаје употреба постојећег језика са паралелним конструкцијама ниског нивоа. MPI, *pthreads* и OpenMP доминирају савременим светом паралелног рачунарства. Добија се прилично висока ефикасност, као и портабилност кода, али по цену нешто тежег кодирања и отклањања грешака.

MPI (*Message Passing Interface*) је стандардизована спецификација за библиотеке које се баве прослеђивањем порука. MPI имплементације су доступне на готово сваком паралелном рачунару, на радним станицама, кластерима, Linux, OSX, Unix и Windows платформама и подржавају C, C++, Fortran, Python и друге програмске језике и платформе. **PVM** (*Parallel Virtual Machine*) је већ превазиђени стандард и данас се ретко користи. Поред приступа комуникацији процеса разменом порука, ту је и комуникација путем дељене меморије. Тад приступ, је додуше, ограничен на појединачне вишепроцесорске системе (*SMP - Symmetric MultiProcessing*). То је једноставније окружење са конкурентним нитима које деле исти адресни простор, па је и програмирање нешто једноставније. OpenMP је најпознатији API за системе дељене меморије.

Најефикаснији приступ је, бар у теорији, коришћење механизма дељене меморије унутар појединачног SMP-а, одн. чвора у кластеру и употреба MPI за комуникацију између различитих чворова у кластеру. У последњих неколико година, употреба масивно паралелних графичких процесора бива све популарнија, и то нарочито у комбинацији са већ стандардним приступима које пружају MPI и OpenMP.

Платформе за развој паралелних програма новијег датума, као што су *Chapel* [13] и *Julia* [14], и поред увођења потпуно нових програмских конструкција и језика за паралелно програмирање, пружају бројне погодности. Највећа од њих тиче се растерећења програмера од бриге о интерпроцесној комуникацији

на најнижем нивоу, која је карактеристична за MPI. Брзо растућа комуна око ових пројекта веома обећава.

Ту су и програмски системи настали у оквиру великих Интернет компанија као што су *Google* и *Facebook*, из потребе да се процесира велика количина података (тзв. *Big Data*), као што су *Apache Hadoop*, *Apache Spark* [15] и њима слични. Ова батерија алата пружа широк подскуп масивно паралелних операција као што су филтрирање, мапирање, редукција исл. Велики број проблема који се срећу у савременој науци (али не сви) могу да се реше помоћу поменутих образца, и то уз толеранцију на грешке и увећану скалабилност у односу на стандардни MPI. Наступају интересантна времена у свету паралелног рачунарства.

Глава 2

Архитектура паралелних рачунара

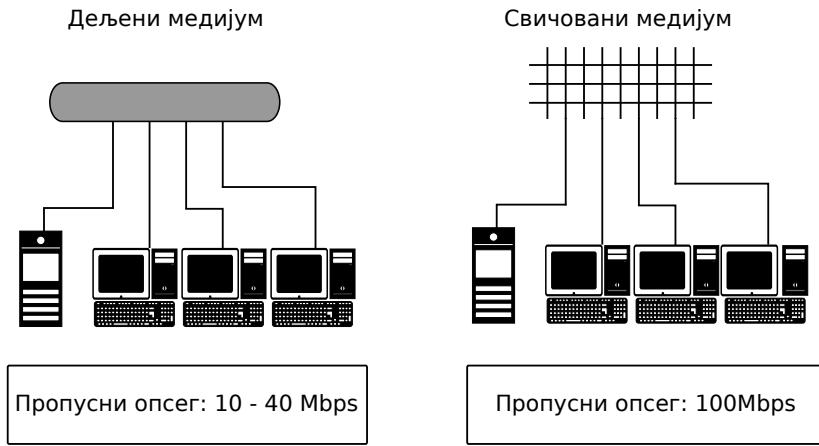
2.1 Архитектура мреже

Од 60-их до средине 90-их година прошлог века, испитане су многе архитектуре паралелних рачунара. Неке компаније су се ослониле на VLSI технологију и развиле сопствене процесоре за своје паралелне рачунаре. Друге су се определиле за широко доступне процесоре опште намене. Основна дебата у то време би се могла формулисати као: **Да ли паралелни рачунар треба да има неколико десетина врло моћних процесора или на хиљаде процесора?** Данас су рачунарски системи са хиљадама примитивних процесора реткост, док специјализовани процесори нису могли да прате брзи напредак процесора широке намене. Зато се данас паралелни рачунари угланом конструишу од уобичајених, тржишно доступних процесора.

Са друге стране, интерконекционе мреже пружају могућност процесорима у два различита рачунара да међусобно комуницирају. У оквиру једног рачунарског система могу се користити и механизми дељене меморије, али је у било ком типу кластера, услед непостојања заједничког адресног простора, размена порука једини могући начин интерпроцесне комуникације.

Дељени медијум, као што је, рецимо, *Ethernet* мрежа изведена коришћењем разводника (*hub*), дозвољава трансфер само једне поруке у једном временском тренутку. Како процесори шаљу поруке преко дељеног медијума, то значи да сви процесори слушају све поруке, а примају само оне намењене њима. Према CSMA (*Carrier Sense Multiple Access*) приступу, процесори пре слања ослушкују да ли је медијум слободан. Ако случајно два или више процесора пошаљу поруке истовремено, долази до колизије, па сваки од процесора чека неки насумични период времена пре поновног слања. Постоје и изведбе CSMA протокола у којима процесори уместо паузирања после колизије емитују не са вероватноћом један,

већ са неком низом вероватноћом [2].



Слика 2.1: Дељени и свичовани медијуми

Свичовани медијум подржава комуникацију типа од-тачке-до-тачке (P2P), дозвољава конкурентну трансмисију више порука између различитих парова процеса и добро подржава скалирање, тј. додавање нових процесора.

Очигледно је да дељени медијум није погодан за интерпроцесорску комуникацију унутар паралелних рачунара. Брзим развојем свичованих мрежних технологија отворен је пут ефикасним и скалабилним паралелним рачунарима. Такође, *Ethernet* као стандард сам по себи није развијан за примену у суперрачунарима, већ за локалне мреже уопште. Такав концепт наметнуо је и нека ограничења, као што је релативно висока латенција, и поред знатно повећаног пропусног опсега, који иде до 10 Gbps . Данас се за супер-рачунаре уместо стандарданог *Ethernet*-а користе неки други мрежни стандарди, од којих је најпознатији *Infiniband*. Латенција код *Infiniband* је врло ниска, чак $0,5\text{ }\mu\text{s}$ са пропусним опсезима од чак 200 Gbps . Поређења ради, латенција стандарданог гигабитног *Ethernet*-а је $50\text{ }\mu\text{s}$, дакле за два реда величине виша.

2.1.1 Топологије супер-рачунарских мрежа

Сваки процесор је повезан на сопствени свич. Свичеви су повезани са процесорима и/или другим свичевима. Код **директне топологије** је однос броја свичева и броја процесора један према један. Код **индиректне топологије** је однос броја свичева и процесора већи од 1:1. Наиме, неки свичеви само повезују друге свичеве.

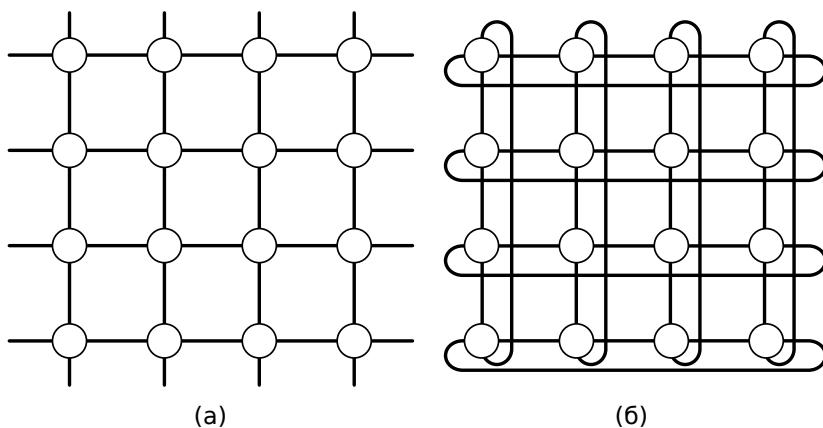
Перформансе дате конфигурације се могу квантификовати користећи неколико параметара. У супер-рачунарству се углавном користе следећи:

- **Дијаметар** је највећа удаљеност између два свича. Низи дијаметар даје боље перформансе.
- **Бисекциона ширина** је минимални број веза између свичева које треба прекинути да би се мрежа поделила на два једнака дела. Виша бисекциона ширина је боља, јер омогућава већи проток података.
- **Број веза по свичу** је пожељно да буде константан и независан од величине мреже, јер се онда мрежа лакше скалира.
- **Дужина ивица** је пожељно да буде константна. Због лакшег скалирања је боље да се чворови и ивице организују у тродимензионом простору и да максимална дужина ивице буде константна и потпуно независна од величине саме мреже.

У наставку ће бити изложено неколико најчешћих топологија које се користе код паралелних рачунара.

2.1.2 Дводимензиона решетка

Дводимензиона решетка је директна топологија у којој су свичеви поређани у решетку (Слика 2.2). Минимални дијаметар и максимална бисекциона ширина постижу се обликом квадрата. Тада су оба ова параметра реда \sqrt{n} за n чворова (процесора).

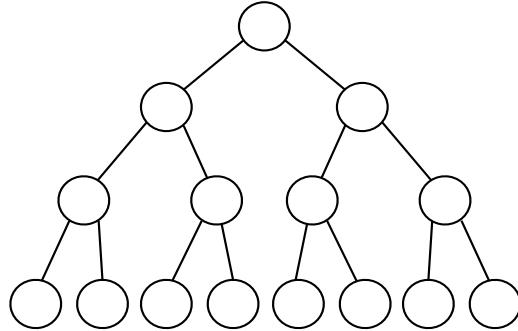


Слика 2.2: Дводимензиона решетка. Лево: Стандардна варијанта. Десно: Обвојна варијанта

Обвојна варијанта дводимензионе мреже има константан број ивица по чвиру. Оно што је такође повољно је чињеница да је могуће направити произвољно велику дводимензиону решетку са константном дужином ивице.

2.1.3 Бинарно стабло

Број процесора у топологији бинарног стабла је 2^d . Број свичева је $2n - 1$ тј. $2 * 2^d - 1 = 2^{d+1} - 1$. На првом нивоу се налази један свич, на дугом два, на трећем четири итд.



Слика 2.3: Бинарно стабло

Ако је са d означенa дубина стабла, онда је број процесора један броју листова стабла, тј. $n = 2^d$. У обрнутом смеру, ако је дат број процесора, дубина стабла се може изразити као $d = \log_2 n$. У циљу израчунавања укупног броја свич чворова, напишимо њихов број по нивоима:

$$\begin{aligned} & 1, 2, 4, 8, \dots \\ & 1, 1 \cdot 2, 1 \cdot 2 \cdot 2, 1 \cdot 2 \cdot 2 \cdot 2, \dots \\ & a_1, a_1 \cdot q, a_1 \cdot q^2, a_1 \cdot q^3, \dots \end{aligned}$$

што чини геометријски низ са параметрима:

$$a_1 = 1 \text{ и } q = 2.$$

Укупан број свичева може се израчунати једноставном применом формуле за суму геометријског низа:

$$\begin{aligned} S_k &= a_1 \cdot \frac{q^k - 1}{q - 1} \\ S_{d+1} &= 1 \cdot \frac{2^{d+1} - 1}{2 - 1} = 2^{d+1} - 1 = 2 \cdot 2^d - 1 = 2n - 1. \end{aligned}$$

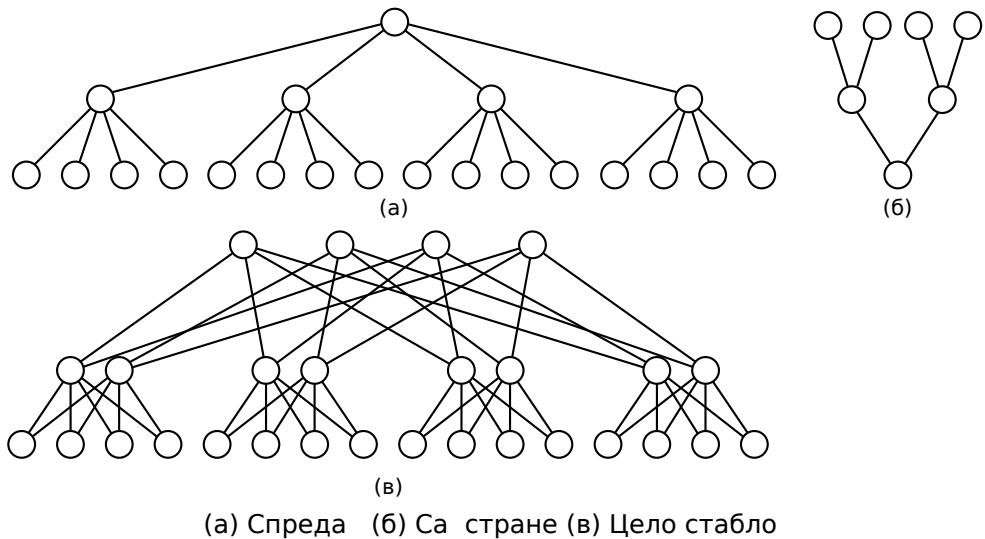
Сваки процесор је повезан на лист бинарног стабла, па је ово индиректна топологија. Карактеристичан за ову топологију је и низак дијаметар, једнак двострукој дубини стабла (од неког листа из леве половине стабла до неког листа из десне

половине стабла) или $2 \log_2 n$. Међутим, бисекциона ширина је минимална могућа - јединична.

Унутрашњи свич чворови имају највише три везе, две за потомке и једну за родитеља. Овај тип топологије је непогодан за постављање у реалну просторију, јер се не пресликава једноставно у тродимензиони простор. Наиме, немогуће је порећати чворове тако да, како се број чворова повећава, дужина најдуже ивице остане увек мања од неке задате константе (карактеристичне димензије просторије).

2.1.4 Хиперстабло

Хиперстабло је индиректна топологија која задржава мали дијаметар бинарног стабла, али има већу бисекциону ширину. Хиперстабло степена k и дубине d са предње стране изгледа као стабло k -тог реда дубине d , док из профила изгледа као обрнуто бинарно стабло дубине d .



Слика 2.4: Хиперстабло за $k = 4$ и $d = 2$

Број процесора у стаблу приказаном на Слици 2.4 је 4^d , тј. k^d . Укупан број свичева је 2^d тј. $2^{d+1} - 1$. Сваки чвор има четворо деце (тј. k деце) и два родитеља (звог обрнутог бинарног стабла). Значи, на сваком следећем нивоу има двоструко више чворова ($\frac{k}{2}$ пута више). На врху стабла имамо 2^d чворова (број листова у обрнутом бинарном стаблу). На првом нивоу имамо 2^d , на другом $2 \cdot 2^d$, на трећем $4 \cdot 2^d$ итд. Укупан број свичева добија се следећим резоновањем:

$$2^d, 2 \cdot 2^d, 2^2 \cdot 2^d, \dots \text{ или } 2^d, 2^d \cdot \frac{k}{2}, 2^d \cdot \left(\frac{k}{2}\right)^2, \dots$$

$a_1, a_1 \cdot q, a_1 \cdot q^2$ - геометријски низ
 $a_1 = 2^d, q = 2$ или $a_1 = 2^d, q = \frac{k}{2}$.

Сума геометријског низа износи:

$$S_n = a_1 \cdot \frac{q^n - 1}{q - 1},$$

$$S_{d+1} = 2^d \cdot \frac{2^{d+1} - 1}{2 - 1} = 2^d \cdot (2^{d+1} - 1),$$

или општије

$$S_{d+1} = 2^d \cdot \frac{(k/2)^d - 1}{k/2 - 1}.$$

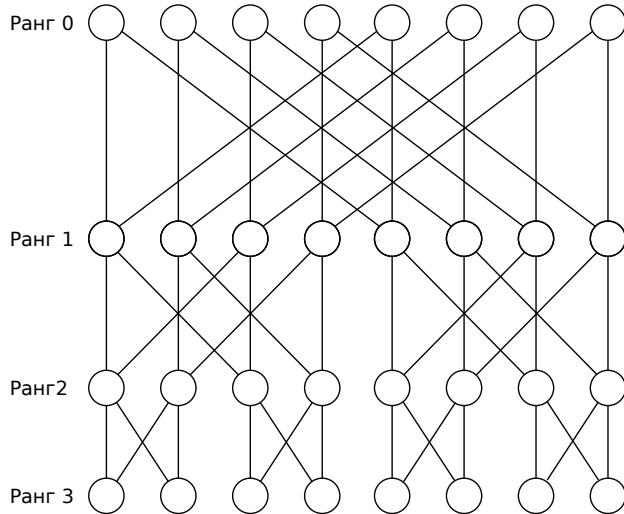
Дијаметар хиперстабла је релативно мали ($2d$), исто као и код бинарног стабла. Бисекциона ширина је, међутим, већа него код бинарног стабла и износи 2^{d+1} . На врху хиперстабла има 2^d чворова и из сваког од њих полазе четири везе, укупно $4 \cdot 2^d$ веза. Од тих веза, половина иде на супротну страну стабла, па стога треба пресећи $\frac{4 \cdot 2^d}{2} = 2 \cdot 2^d = 2^{d+1}$ ивица да би се мрежа преполовила. У општем случају, бисекциона ширина је $\frac{k \cdot 2^d}{2} = k \cdot 2^{d-1}$. Број ивица по свич чвиру ниkad није већи од шест (два родитеља и четворо потомака). Максимална дужина ивице је, међутим, растућа функција величине мреже.

2.1.5 Топологија лептира

Топологија лептира је индиректна топологија. Број процесора у овој врсти мреже је 2^d , где је d број тзв. рангова.

Број свич чворова у овој мрежи је $n(\log_2 n + 1)$. Свичеви су поређани у n колона (сваки за по један процесор) и $\log_2 n + 1$ врста или рангова. $\log_2 n$ је дубина, зато што се нпр. осам чворова дели у четири, четири се деле у по два, два се деле у по један. Ако је $\log_2 n$ дубина, то значи да од врха до дна има $\log_2 n$ веза које повезују $\log_2 n + 1$ чворова, одакле долази $\log_2 n + 1$ рангова. Рангови су нумерисани од 0 до $\log_2 n$. Претпоставка архитектуре је да су чворови ранга 0 и ранга $\log_2 n$ у ствари исти чворови.

Ако је свич обележен са (i, j) , то значи да је у питању j -ти свич i -тог ранга. Свич (i, j) је повезан са два чвора ранга $i - 1$, и то $(i - 1, j)$ и $(i - 1, m)$, где је m



Слика 2.5: Топологија лептира

додијено инверзијом i -тог бита по важности у бинарној репрезентацији j . Нпр. $(2,3) - 011$ је повезано са $(1,3)$ и $(1,1)$, јер $011 \rightarrow 001$ за $i = 2$.

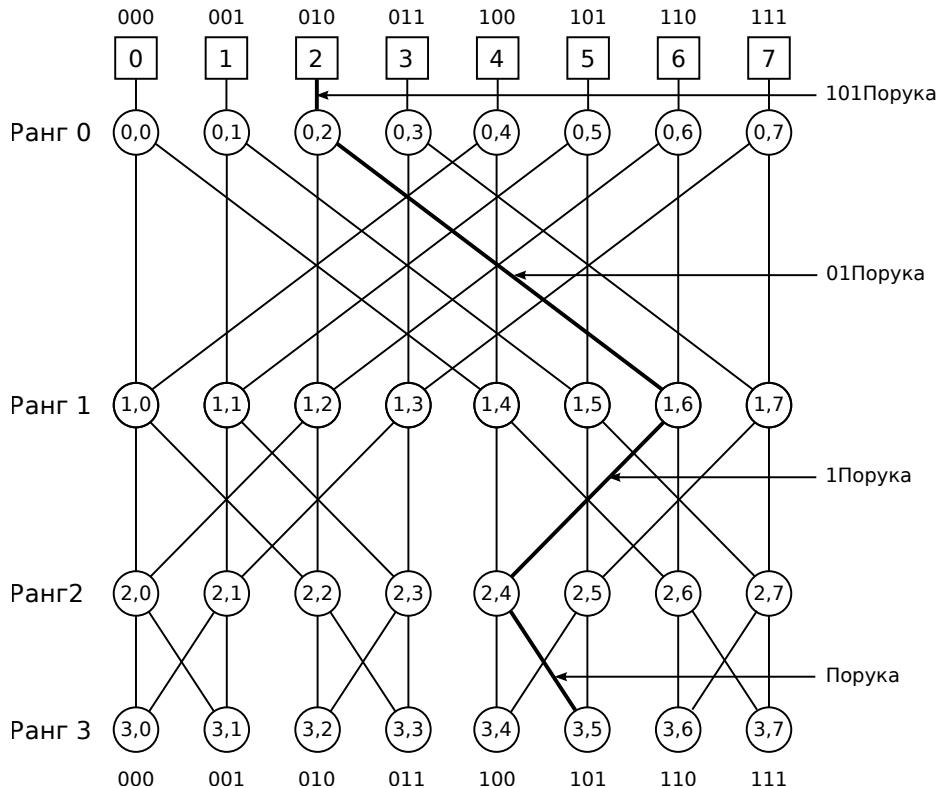
Дијаметар мреже је $\log_2 n$, док је бисекциона ширина једнака n . Како сваки чврт има два потомка, из n чворова 0-тог ранга полази $2n$ веза, а од тога n иде у исту половину мреже, а других n у супротну. Тих n веза треба пресећи да би се мрежа пресекла на два једнака дела.

Сваки свич чврт има константан број од четири везе. С друге стране, најдужа ивица се повећава како расте број чворова, јер неке ивице два суседна ранга морају да пређу пут од половине свичева да би стигле на другу половину.

Алгоритам за рутирање порука функционише по следећем једноставном принципу:

- Сваки свич чврт узима водећи бит из поруке.
- Ако је тај бит нула, шаље остале битове доле лево, ако је тај бит један, доле десно.

Порука је, у ствари, бинарна ознака редног броја одредишног процесора. Речимо, ознаком 101 из било ког процесора стижемо до процесора за ознаком 5 (Слика 2.6). На првом нивоу, све везе налево воде у колоне чији бинарни број почиње са 0, а све везе надесно у колоне које почињу са 1. На другом нивоу, лево воде у колоне чији је други бинарни број нула а, десно чији је други бит један итд.



Слика 2.6: Рутирање у топологији лептира

2.1.6 Хиперкоцка

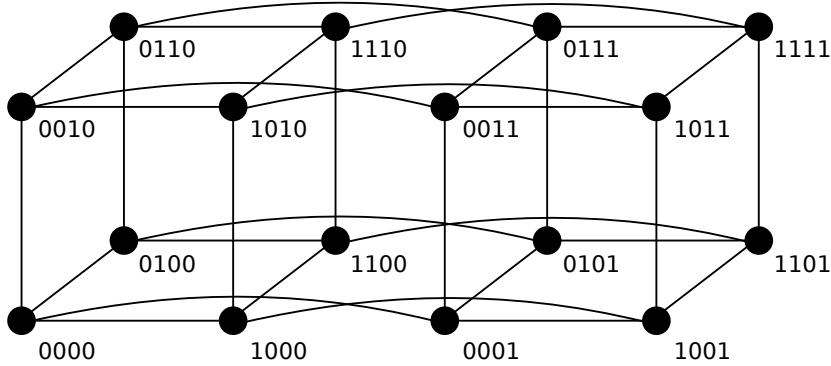
Хиперкоцка, у ствари, представља топологију лептира у којој је свака колона свич чврова колабирана у један чвр. Тиме су избрисане везе које не мењају бит, а сваки чвр има везу која мења први бит, везу која мења други бит, и тако до $\log_2 n$. Пошто је код лептира свака колона била везана за по један процесор, овде је сваки чвр везан за по један процесор, па је топологија, у ствари, директна. Дакле, број процесора у хиперкоцки је 2^d , као и број свичева.

Процесори су нумерисани од 0 до $2^d - 1$. У бинарном запису имају d цифара. Дакле:

$$\begin{array}{lllll} 0001 & 0010 & 0100 & 1000 & 10000-1=1111 \\ 2^0 = 1 & 2^1 = 2 & 2^2 = 4 & 2^3 = 8 & 2^4 - 1 = 15 \end{array} \quad d = 4 = \log_2 16.$$

Свич чврлови су суседни ако се разликују за тачно један бит. Дијаметар је $\log_2 n$, јер је толики број бинарних цифара ознаке једног процеса, тј. толико корака је потребно да се промене сви битови. Бисекциона ширина хиперкоцке је

$\frac{n}{2}$. Кад спојимо два дводимензионална квадрата повезивањем одговарајућих темена, добијемо тродимензиону коцку. Ако узмемо две тродимензионе коцке и спојимо одговарајућа темена, добијамо четвородимензиону хиперкоцку, приказану на Слици 2.7.



Слика 2.7: Четвородимензиона хиперкоцка

У дужину се мења први бут отпозади, у висину други, у ширину трећи и у четврту димензију последњи. Имамо укупно n чвррова. У датом примеру обе тродимензионе коцке имају по $\frac{n}{2}$ чвррова. Тих $\frac{n}{2}$ парова одговарајућих чвррова су међусобно повезани, и то су везе које треба пресећи да би се преполовила мрежа (бисекциона ширина).

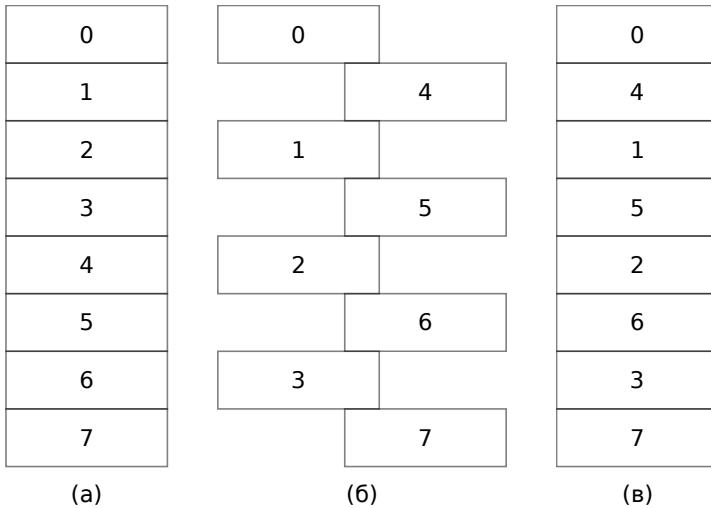
Дијаметар и бисекциона ширина су веома повољни, али на уштрб осталих параметара. Број ивица по чврву је $\log_2 n$. Сваки чврв има бинарну дужину $\log_2 n$. За сваки бит који може да се промени, свич чврв поседује по једну везу, што до води до отежаног скалирања. Такође, дужина најдуже ивице расте са повећањем мреже.

Рутирање порука у хиперкоцки одређује чињеница да број битова у којима се ознаке свичева разликују одређује њихово растојање. На пример, од процесора 0101 (5) до процесора (3) растојање је једнако 2:

$$\begin{aligned} 0101 &\rightarrow 0001 \rightarrow 0011 \\ &\text{или} \\ 0101 &\rightarrow 0111 \rightarrow 0011. \end{aligned}$$

2.1.7 *Shuffle exchange* топологија

Идеја за ову врсту топологије је савршено измешан шпил карата, као што је приказано на Слици 2.8.



Слика 2.8: Савршено измешан шпил карата

Дакле, као што се види са Слике 2.8, да би се добила позиција карте у измешаном шпилу, врши се померање битова уз премештање првог бита слева на крај (ротација). На пример:

$$\begin{array}{llll} 000 \rightarrow 000 & 100 \rightarrow 001 & 001 \rightarrow 010 & 101 \rightarrow 011 \dots \\ 0 \rightarrow 0 & 4 \rightarrow 1 & 1 \rightarrow 2 & 5 \rightarrow 3 \end{array}$$

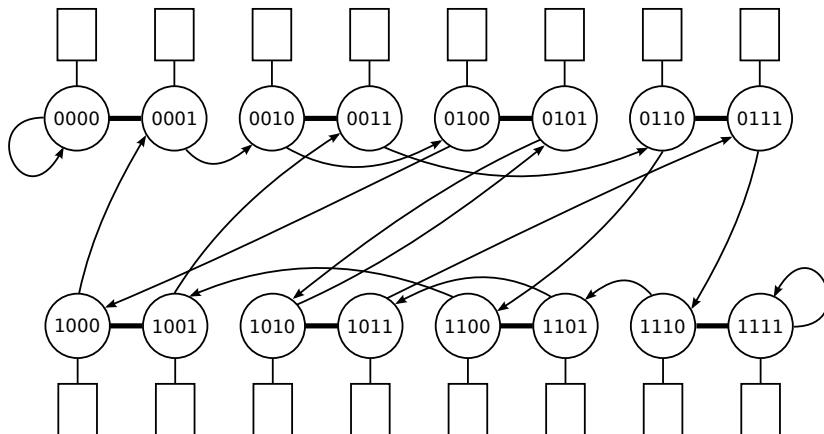
Ово је пример директне топологије. Број процесора је $n = 2^d$, док је број свичева исти. Нумеришу се од 0 до $n - 1$. У *shuffle exchange* архитектури постоје две врсте веза:

- **Exchange везе** - повезују свичеве чије су бинарне ознаке разликују у најмање значајном биту. Ове везе су двосмерне.
- **Shuffle везе** - повезују свич i са свичом j , где се j добија цикличном битовском ротацијом бинарног записа броја i . Ове везе су једносмерне.

Дијаметар за мрежу од n процесора је $2 \log_2 n - 1$. Чворови са највећом удаљеностшћу су 00..00 и 11..11. Да би се дошло од једног до другог потребно је $\log_2 n$ инвертовања (*exchange*) и $\log_2 n - 1$ ротирања (*shuffle*), што укупно даје $2 \log_2 n - 1$ корака:

$$\begin{aligned} 0000 &\xrightarrow{E} 0001 \xrightarrow{S} 0010 \xrightarrow{E} 0011 \xrightarrow{S} 0110 \xrightarrow{E} 0111 \xrightarrow{S} 1110 \xrightarrow{E} 1111 \\ n &= 16 \\ E &= 4 = \log_2 n \\ S &= 3 = \log_2 n - 1. \end{aligned}$$

Бисекциона ширина за *shuffle exchange* топологију износи $\frac{n}{\log_2 n}$. Карактеристи-



Слика 2.9: Мрежа *shuffle exchange* са 16 чврата. Дебеле линије представљају двосмерне *exchange* везе, а стрелице једносмерне *shuffle* везе.

чан је и константан број ивица по чврту - две долазеће и две одлазеће. Неповољно утиче само чињеница да дужина најдуже ивице расте са величином мреже, што отежава скалирање.

Закључак

После свих изложених чињеница, може се закључити да ни једна мрежа није оптимална у сваком погледу. Све имају логаритамски дијаметар осим дводимензионе решетке. Хиперстабло, лептир и хиперкоцка имају високу бисекциону ширину. Све имају константан број ивица по чврту осим хиперкоцке. Само дводимензионе решетке одржава дужине ивица константним како се величина мреже повећава. *Shuffle exchange* представља релативно добар компромис, јер има константан број ивица по чврту, низак дијаметар и високу бисекциону ширину.

2.2 Векторски рачунари

Векторски рачунар је рачунар који поред операција са скаларима имплементира и векторске операције. Два су основна начина имплементације векторског рачунара:

1. **Векторски процесор заснован на принципу цевовода.** Повлачи векторе из меморије у процесор, где њима манипулишу аритметичке јединице

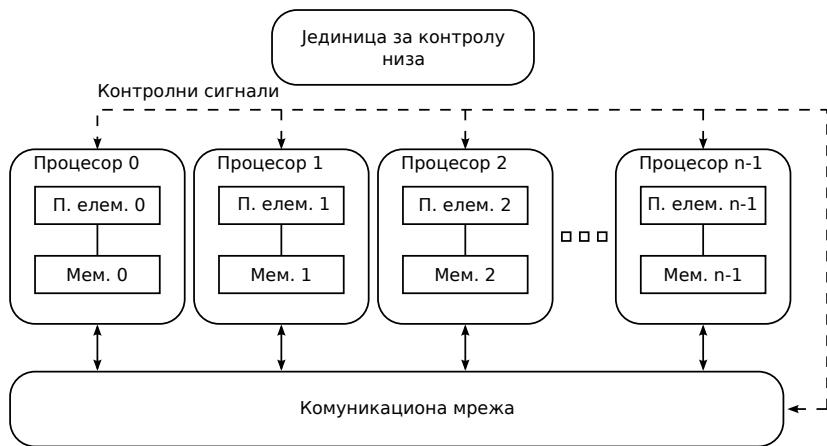
изведене као цевовод. Ово су ране архитектуре и више се активно не развијају.

2. **Процесорски низ.** Чини га секвенцијални рачунар повезан на скуп идентичних, синхронизованих процесорских елемената који могу симултанско да изводе једну исту операцију над различитим подацима. Овај приступ је погодан за примену код проблема са инхерентном паралелизацијом података, као што су нпр. разни типови научних прорачуна. Други мотив за развој приступа процесорског низа био је историјски висока цена управљачке јединице процесора.

У даљем тексту ће бити речи искључиво о архитектури заснованој на процесорским низовима.

Архитектура

Главни процесор (*front-end*) контролише једноставне процесоре у низу (*back-end*) процесоре, као на Слици 2.10. Примарна меморија главног процесора садржи инструкције које треба да се изврше и податке којима треба манипулисати и функционише у секвенцијалном маниру. Процесорски низ се, у ствари, састоји се из више парова процесор-меморија. Подаци којима се манипулише паралелно су дистрибуирани међу овим меморијама. У сваком циклусу, главни процесор обазнајује паралелну инструкцију процесорима у процесорском низу. Процесори симултанско извршавају ту инструкцију над операндима у њиховим локалним меморијама.



Слика 2.10: Процесорски низ

Ако за пример узмемо операцију сабирања, и ако је број елемената вектора мањи или једнак од броја процесора, сабирање вектора може да се изврши у једној јединој инструкцији.

Пример 2.1. На 1024 процесора, сабирање два броја траје 1 ns . Сабирање два вектора од 1024 члана може се обавити уз следеће перформансе:

$$P = \frac{1024\text{ ops}}{1\text{ ns}} = \frac{1024\text{ ops}}{10^{-9}\text{ s}} = 1024 \cdot 10^9 \text{ ops/s}.$$

Пример 2.2. Низ чини 512 процесора, а сабирање два броја траје 1 ns . Сабирање два вектора од по 600 чланова даје следеће перформансе:

$$P = \frac{600\text{ ops}}{2\text{ ns}} = 3 \cdot 10^8 \text{ ops/s}.$$

Перформанса је мера која показује колико посла може да се заврши у јединици времена. Јединица је операција у секунди. Као што се из ова два примера може видети, перформанса је највиша кад су сви процесори активни и када је количина података дељива бројем процесора.

Комуникациона мрежа процесорског низа је такође битан чинилац перформанси. Наиме, типична векторска операција је много компликованија од простог сабирања. Рецимо, операција попут

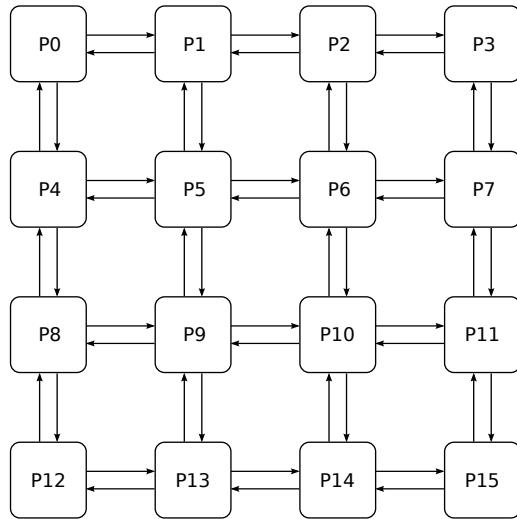
$$a_i = \frac{a_{i-1} + a_{i+1}}{2}$$

је веома честа у моделирању физичких и инжењерских проблема. Овде су процесору i потребне вредности из меморија других процесора, што имплицира да мора да постоји интерпроцесорска комуникација. За процесорски низ се најчешће користи 2Д решетка (Слика 2.11) зато што се она лако имплементира у VLSI и зато што подржава конкурентни трансфер порука.

Појава **гранања** доводи до значајног пада перформанси. Овај проблем биће објашњен на примеру. Рецимо да вектор има од 10 елемената дистрибуираних на 10 процесора. Задатак који треба извршити је да ако је елемент различит од нуле, мења се у 1, а ако је једнак нули, мења се у -1.

Процесорски низ омогућава да само подскуп процесора изврши инструкцију уз помоћ тзв. маскирајућег бита. Прво се провери који елементи су 0 и на њима се подеси маскирајући бит да се не извршавају. Онда се пусти инструкција, а изврше је само они претходно неактивни. На крају се сви маскирајући битови избришу.

Проблем је што се код `if-then-else` конструкције пролази кроз различите гране секвенцијално. Поред тога, као додатни вишак се укључују и искључују процесори, што у глобалу доводи до значајног пада перформанси.



Слика 2.11: Процесори распоређени у дводимензиону решетку која омогућава конкурентну интерпроцесорску комуникацију

Недостаци

Процесорски низови имају неколико значајних недостатака због чега нису у широкој употреби:

1. Немају сви проблеми паралелност података.
2. Пошто у једном тренутку може да се изврши само једна инструкција, гранања успоравају извршавање.
3. Не адаптирају се добро на системе са више корисника.
4. Потребна је добра и скупа мрежа. То не мења много цену ако имамо много процесора, али код мањих система пада однос цена/перформансе.
5. Базирају се на специјално дизајнираним VLSI процесорима, који не могу да прате раст перформанси и пад цена уобичајених процесора.
6. Мотивација за процесорским низовима, висока цена контролних јединица, више не постоји, јер им је цена у доброј мери пала.

И поред наведених мана, форме векторских процесора, поједине форме векторских процесора су се пробиле у свет савременог рачунарства високих перформанси. То су, пре свега, графички процесори опште намене.

2.3 Мултипроцесори

Мултипроцесор је систем са више процесора и дељеном меморијом. Иста адреса на више процесора реферише на исти податак. Мултипроцесори избегавају три проблема процесорских низова:

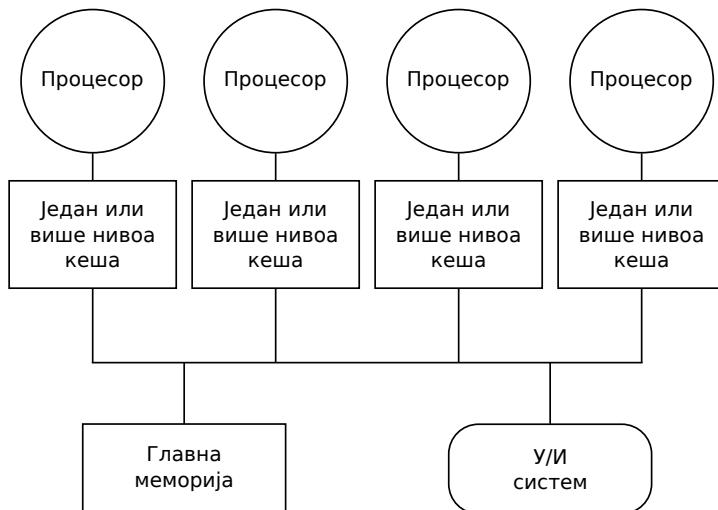
1. Могу се изградити од стандардних процесорских јединица.
2. Природно подржавају више корисника.
3. Не губе ефикасност када налажу на кондиционални паралелни код.

Мултипроцесори се деле на **централизоване и дистрибуиране мултипроцесоре**. Централизованима је сва примарна меморија на једном месту, а дистрибуираним раздељена међу процесорима, али и даље чини јединствени адресни простор.

Типични унипроцесор користи магистралу да повеже процесор са примарном меморијом У/И уређајима. Кеш меморија смањује фреквенцију којом процесор мора да чека док му се доведу подаци из примарне меморије.

2.3.1 Централизовани мултипроцесор

Централизовани мултипроцесор је јасна екstenзија унипроцесора. Додатни процесори се повезују на магистралу и сви деле заједничку радну меморију.



Слика 2.12: Централизовани мултипроцесор

Перформансе приступа меморији су исте за све процесоре, и зато се ова архитектура назива *Uniform Memory Access* (UMA) или *Symmetric Multi Processor* (SMP). У складу са архитектуром, подаци могу бити двојаки:

- **Приватни подаци** - приступа им само један процесор.
- **Дељени подаци** - користи их више процесора.

У централизованом мултипроцесору, процесори комуницирају коришћењем дељених података. Проблеми са дељеним подацима су кеш кохернција и синхронизација. Савремени персонални рачунари, па чак и велики број табличних рачунара и паметних телефона су засновани на SMP архитектури. Захваљујући високом степену интеграције, данас је могуће више процесорских језгара сместити у једноједином интегралном колу.

Кеш кохеренција - репликација података на више кеш меморија редукује перформансе. Питање је како се осигурати да сви процесори имају исти податак на истој меморијској адреси. Један процесор промени неку вредност, а други у свом кешу и даље има стару вредност.

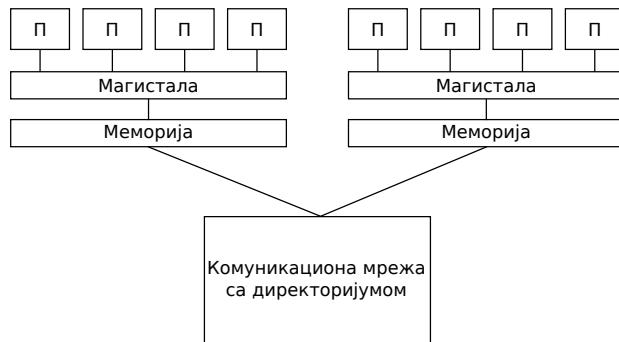
***Snooping* протокол** је једно од хардверских решења за проблем кеш кохеренције. Сваки кеш контролер ослушкује који се кеш блокови позивају од стране других процесора. Ако неки процесор хоће да упише нову вредност, све кеширане копије тог податка код других процесора постају невалидне и покушај приступа њима доводи до кеш промашаја. Поједини аутори овај алгоритам називају још и *write invalidate* протоколом.

Синхронизација процесора користи два механизма који су познати и са једнопроцесорских система. **Mutex** (*mutual exclusion*) омогућава да само један процес може да буде умешан у одређену специфицирану активност у једном тренутку. **Баријера** имплицира да ниједан процес неће прећи одређену тачку у програму звану баријером, док сви процеси не стигну до те тачке.

2.3.2 Дистрибуирани мултипроцесор

Пропусна моћ магистрале ограничава SMP на неколико десетина процесора. Алтернатива је дистрибуирати примарну меморију међу процесорима. Ту је локални приступ меморији много бржи него нелокални, Слика 2.13. Због просторне и временске локалности унутар програма, могуће је готово све уредити тако да се углавном приступа локалној меморији. Зато дистрибуирани мултипроцесори имају већи проток података и мање време приступа меморији него централизовани, а допуштају и већи број процесора.

Процесори у дистрибуираним мултипроцесорима имају **заједнички адресни простор** упркос дистрибуиранијој меморији. То значи да иста адреса на различитим процесорима реферише на исту меморијску локацију. Ови системи се



Слика 2.13: Шема дистрибуираног мултипроцесора

још зову *Non Uniform Memory Access* (NUMA), јер време приступа меморији варира у зависности од тога да ли је референцирана адреса у локалној меморији тог или неког другог процесора.

2.4 Мултикомпјутер

Мултикомпјутери су системи са дистрибуираном меморијом и више процесора, али за разлику од NUMA мултипроцесора, **процесори не деле адресни простор**. Сваки процесор има приступ само својој локалној меморији. Иста адреса на различитим процесорима означава различите податке. Процесори комуницирају разменом порука, и не постоји проблем кеш кохеренције.

Комерцијални мултикомпјутери имају балансиране брзине процесора и брзину мреже. Мреже су свичоване, имају мало кашњење и велики проток. Тзв. *Beowulf* кластери, за разлику од њих, користе рачунаре и свичеве масовне производње. То је значајно јефтинији систем, али је латенција порука доста већа, а проток нешто мањи.

2.4.1 Асиметрични мултикомпјутер

По идеји је сличан процесорском низу. Главни рачунар комуницира са корисницима и У/И уређајима, а процесори у позадини извршавају само паралелне послове.

Предности: Пошто ништа друго не окупира позадинске процесоре, лакше је разумети, моделовати паралелну апликацију и побољшавати њене перформансе. Овим процесорима је довољан примитивни оперативни систем, једноставан за повезивање.

Мане: Ако главни рачунар откаже, отказује цео систем. Такође, један једини главни рачунар ограничава скалабилност система, тј. ограничен је број кори-

сника које може да опслужи, и за то време позадински процесори не раде ништа. Поред тога, примитивни оперативни систем на позадинским рачунарима отежава отклањање грешака, јер не омогућава У/И операције, па ови процесори, на пример, не могу на једноставан начин да штампају поруку о грешци. Даље, свака апликација захтева развијање два програма. *Front-end* програм интерагује са корисницима и фајл-системом, шаље податке *back-end* процесорима и прима резултате од њих. *Back-end* програм је одговоран за делове алгоритма са интезивним рачунањем.

2.4.2 Симетрични мултикомпјутер

Сваки компјутер има исти оперативни систем и исте функционалности. Сви рачунари могу да извршавају исти паралелни програм.

Предности: Нема загушења главног рачунара. Ако је неки рачунар преоптерећен, корисници могу да се пријаве на неки други. Подршка за отклањање грешака је боља јер сваки процесор може да штампа поруку о грешци. Нема више раздвојених *front-end* и *back-end* програма, јер сваки процесор извршава исти програм.

Мане: Теже је одржати илузију јединственог паралелног рачунара. Нема лаког начина да се балансира развој програма (кад корисник употребљава рачунар). Теже је остварити високе перформансе када има више процеса на истом процесору и када се они такмиче за процесорске циклусе, простор у кешу и меморијску магистралу.

Интересантан је модел изложен као спој симетричног и асиметричног (ParPar кластер, [4]). Сваки рачунар има исти оперативни систем (симетрично). Сваки процесор извршава само један процес, па на неке не може увек да се логује (асиметрично).

2.5 *Beowulf (Commodity)* кластери

Углавном су састављени од рачунара и свичева масовне производње, који се сви налазе на истој физичкој локацији, једни до других, Слика 2.14. Ево неколико карактеристика оваквих система:

- Посвећени су извешавању паралелних послова.
- Рачунарима може да се приступи само са удаљене локације преко мреже, нема тастатуре нити екрана.
- Сви рачунари имају исти оперативни систем и идентичну слику (*image*) оперативног система на локалном диску.
- Цео кластер се администрира као јединствени ентитет.

Оваквом систему се може супротставити **мрежа радних станица**, јер се, у принципу, ради о сличном хардверу. Међутим, мрежа радних станица показује следеће непогодности:

- Састављена је од просторно расутих рачунара који се обично налазе у канцеларијама.
- Приоритет су потребе корисника, док се паралелни програми извршавају у позадини.
- Различите радне станице могу имати различите оперативне системе и различите локалне дискове.
- Корисници могу да искључе и рестартују радне станице, па постоји потреба за имплементацијом механизама контролних тачака и рестартовања паралелних послова.



Слика 2.14: Лево: *Beowulf* кластер. Десно: Суперкомпјутер

2.6 Флинова таксономија

Флинова таксономија даје засад најбољу класификацију паралелних рачунара. Категорија зависи од паралелизације тока инструкција и тока података. Процес се посматра као секвенца инструкција (ток инструкција) којима манипулише секвенца података (ток података). Нагласак је на хардверу који манипулише овим токовима.

Хардвер може да подржава један ток инструкција или вишеструки ток инструкција који манипулише једним током података или вишеструким током података. Одатле четири категорије:

SISD (Single Instruction Single Data) - један процесор оперише на једном току података. Не рачунају се копроцесори за нпр. У/И операције које ипак могу да уведу неку конкурентност. Али, то није конкурентност у извршавању програма, већ у процесирању унутар рачунара.

SIMD (Single Instruction Multiple Data) - иста инструкција се симултано извршава над различитим деловима података. Примери су векторски процесори типа цевовода и процесорски низови, а у новије време графички процесори опште намене.

MISD (Multiple Instruction Single Data) - цевовод више независних функционалних извршних јединица које оперишу над једним током података и проследију резултате од једне функционалне јединице до друге. Пример је тзв. системни низ [5], али комерцијално није значајно заступљен.

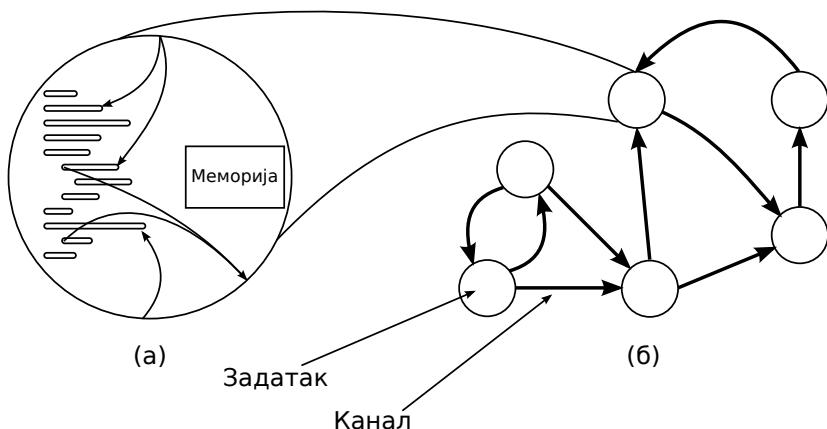
MIMD (Multiple Instruction Multiple Data) - Рачунари са вишеструким процесорима, где различити процесори могу симултано да извршавају различите токове инструкција над различитим токовима података. Примери су мултипроцесори и мултирачунари. Већина савремених паралелних рачунара спада у MIMD категорију.

Глава 3

Пројектовање паралелних алгоритама

3.1 Модел задатак/канал

Модел задатак/канал представља паралелно извршавање као скуп задатака који интерагују слањем порука кроз канале. **Задатак** чине програм, његова локална меморија и скуп У/И портова. **Канал** је ред за поруке који повезује излазни порт једног задатка са улазним портом другог задатка, као на Слици 3.1.



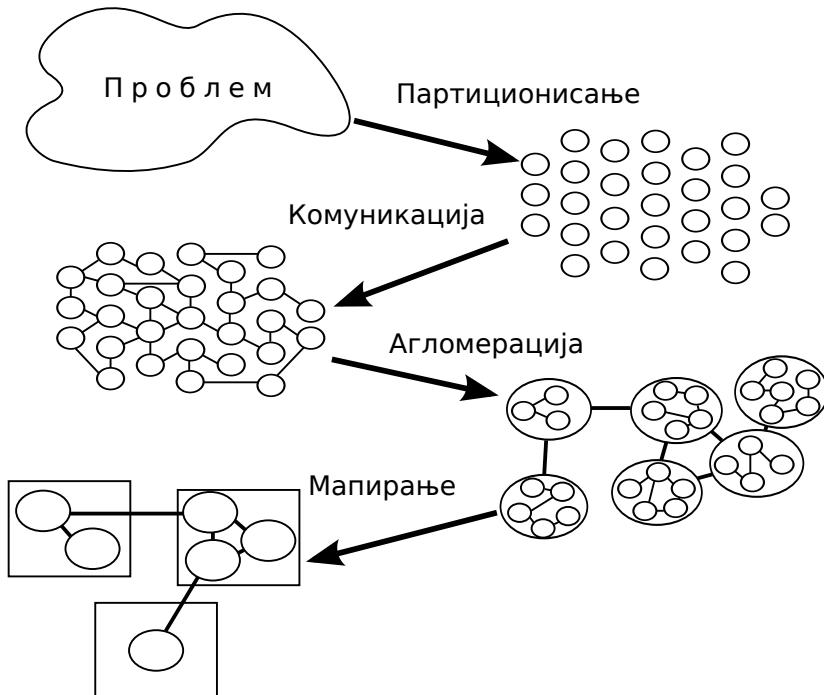
Слика 3.1: Модел задатак/канал

Примање порука је блокирајућег карактера. Ако неки задатак покуша да прими вредност која још није доступна, онда мора да чека. Слање порука никад не блокира задатак, чак и ако у тренутку слanja нико не покушава да прими поруку. Дакле, примање је синхронна операција, док је сланај асинхронна операција.

3.2 Фостерова методологија

Фостерова методологија дизајн паралелног алгоритма подразумева процес од четири корака:

1. Партиционисање
2. Комуникација
3. Агломерација
4. Мапирање



Слика 3.2: Фостерова методологија

3.2.1 Партиционисање

Партиционисање подразумева процес дељења прорачуна и података на делове. Добро партиционисање раздваја и израчунавање и податке на велики број сасвим малих делова. Може се говорити о два приступа партиционисању:

- **Декомпозиција домена** је приступ у којем делимо податке и одређујемо како да повежемо прорачуне са подацима. Примери за овај приступ су подела низа на поднизове, подела петље на потпетље, подела простора на потпросторе.
- **Функционална декомпозиција** је приступ у којем прво делимо функције израчунавања на мање делове, а онда одређујемо како да повежемо податке са израчунавањима. На овај начин се обично добије колекција задатака, који конкурентност постижу механизмом цевовода.

Какву год декомпозицију да одаберемо, сваки новодобијени део називамо **примитивним задатком**. Што више примитивних задатака, то боље, јер је њихов број горња граница паралелизма који се може постићи за дати проблем. Критеријуми квалитета извршеног партиционисања се могу сажети у неколико ставки:

- Примитивних задатака треба да је бар 10 пута више (бар један ред величине) него процесора у циљаном паралелном рачунару или кластеру.
- Редундантна израчунавања и редундантне структуре података су минимизоване.
- Примитивни задаци су углавном исте величине због балансирања међу процесорима.
- Број задатака је растућа функција величине проблема, због пожељне скалабилности.

3.2.2 Комуникација

Одређује вредности које се прослеђују међу формираним примитивним задацима. Две су примарне врсте комуникације:

- **Локална комуникација** - када су задатку потребне вредности из малог броја других примитивних задатака. Креирајмо канале који указују на ток тех података.
- **Глобална комуникација** - када значајни број примитивних задатака мора да пошаље податке да би се извршило неко израчунавање. На пример, треба израчунати суму свих вредности које држе примитивни задаци. Ови канали се не цртају у раној фази дизајна.

Комуникација представља додатни терет паралелног алгоритма, јер она не постоји у секвенцијалном алгоритму. Минимизација овог додатног оптерећења је један од најважанијих циљева ефикасног паралелног програмирања. Критеријум за добро извршен корак одређивања комуникације:

- Комуникационе операције су балансиране међу задацима.
- Сваки задатак комуницира само са малим бројем својих суседа.
- Задаци могу да изводе своје комуникације истовремено.
- Задаци могу да изводе своја израчунавања истовремено.

Прва два корака идентификују што је могуће више паралелизма. Друга два корака узимају у обзир циљану архитектуру.

3.2.3 Агломерација

Ако је број здатака за неколико редова величине већи од броја процесора, само њихово креирање изазива велико додатно оптерећење. Алгомерација је процес груписања задатака у веће задатке, и то са следећим мотивима:

- Побољати перформансе
- Упростити кодирање
- У MPI програмирању, циљ је често да се креира по један алгомеризован задатак по процесору. У овом случају, мапирање је тривијално.
- Смањивање додатног трошка комуникација. Можемо да повећамо локалност тј. да потпуно елиминишемо неку комуникацију тако што ћемо прimitивне задатке који комуницирају спојити у један задатак, или шематски:

$$\circ \rightarrow \circ \Rightarrow \bigcirc$$

Друга могућност је да спојимо задатке пошиљаоце са једне стране и задатке примаоце са друге стране. Овако се смањује број порука које се шаљу. Услед карактеристика мрежног хардвера, краће траје слање једне дуге поруке него слање више кратких порука исте укупне величине. Разлог за овакво понашање је **латенција** (кашњење поруке) која се дефинише као време потребно да се започне слање поруке. Латенција не зависи од дужине поруке. Шематски се овај приступ може приказати као:

$$\begin{array}{c} \circ \rightarrow \circ \\ \circ \rightarrow \circ \end{array} \Rightarrow \bigcirc \rightarrow \bigcirc$$

- Одржавање скалабилности паралелног дизајна. Морамо да се осигурамо да нисмо груписали толики број задатака и на тај начин лимитирали могућност портовања програма на рачунар са више процесора.

Критеријуми за добро урађену агломерацију су следећи:

- Локалност паралелног алгоритма је порасла.
- Поновљене операције троше мање времена него комуникације које замењују.
- Подаци се понављају у доволно малој мери да омогућавају скалирање алгоритма.
- Агломеризовани задаци имају сличне трошкове израчунавања и комуникације.
- Број задатака је растућа функција величине проблема.
- Број задатака је што је могуће мањи, али барем онолики колики је и број процесора на циљном паралелном рачунару.
- Направљен је компромис између агломерације и количине модификација на секвенцијалном коду.

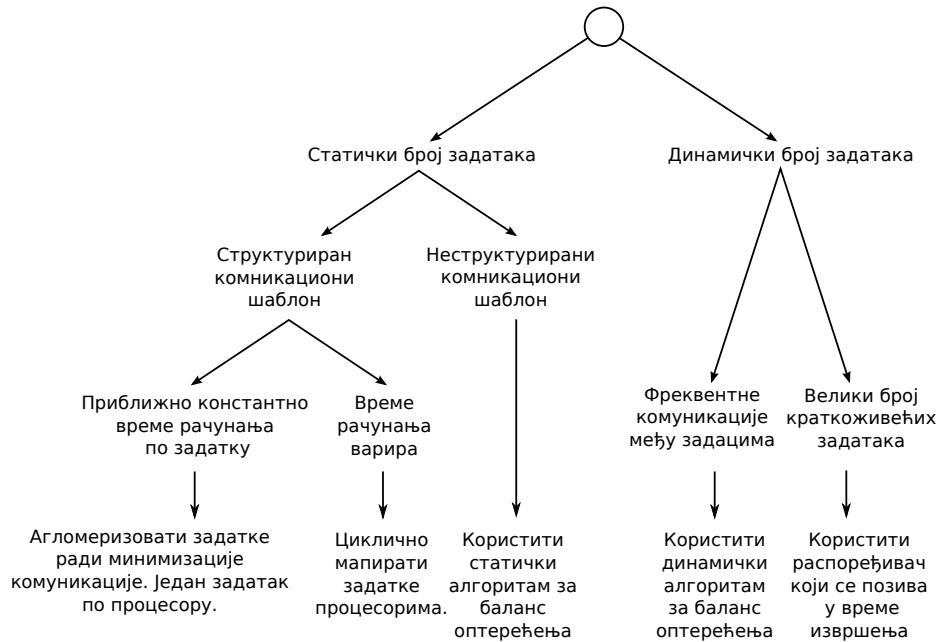
3.2.4 Мапирање

Мапирање је процес додељивања агломеризованих задатака процесорима. Код централизованих мултипроцесора, ово мапирање врши сам оперативни систем. Код система са дистрибуираном меморијом, мапирање врши корисник паралелног програма. Основни циљеви мапирања су да се максимизује искоришћење процесора и да се минимизује међупроцесна комуникација.

Искоришћење процесора је просечан проценат времена током којег су процесори активно извршавали задатке везане за један проблем. Најбоље је кад је посао једнако балансиран и сви процесори почињу и завршавају извршавање у исто време. Супротно томе, није ефикасно када један процесор стоји док остали раде.

Међупроцесна комуникација расте када су два задатка повезана каналом мапирани на различите процесоре. Међупроцесна комуникација опада када су задаци повезани каналом мапирани на исти процесор.

Анализа показује да су ова два циља конфликтна. Рецимо да имамо p процесора. Ако све задатке мапиратмо на исти процесор, комуникација је нулта, али је искоришћење процесора $1/p$. Зато је циљ наћи добар баланс између максимизације икоришћења и минимизације комуникације. Међутим, проналажење оптималног мапирања је NP - комплексан проблем. Зато морамо да се ослонимо на хеуристику, како бисмо овај проблем решили у коначном времену.



3.3 Стабло одлучивања за мапирање задатака на процесоре

Када је проблем партиционисан доменском декомпозицијом, задаци су после агломерације обично сличне величине, тј. добро балансирани. Ако је комуникациони шема структурирана, добра стратегија је креирати *p* агломеризованих задатака који минимизују комуникацију и мапирати сваки задатак на по један процесор.

Када је број задатака фиксан, комуникациони шема структурирана, али се времена извршавања задатака значајно разликују, онда циклично мапирање задатака на пороцесоре добро балансира оптерећење, али уз повећане комуникације.

Ако комуникациони шема није структурирана, онда се користи **статички алгоритам за баланс оптерећења**. Он се покреће пре програма и унапред одређује стратегију мапирања, тако да минимизује додатне комуникационе трошкове.

Ако се задаци креирају и уништавају током временена извршења програма, користити се **динамички алгоритам за баланс оптерећења**. Он се покреће по времену током извршавања програма, анализира тренутно активне задатке и као излаз даје новомапиране задатке на процесоре.

Ако задаци извршавају одређену функцију (потпроблем) свако за себе и не-

ма комуникације међу њима, користи се **алгоритам за распоређивање**. Код **централизованог распоређивача**, процеси су подељени на једног господара и велики број слугу. Господар одржава листу задатака које треба обавити. Када слуга нема шта да ради, врши потраживање посла од господара. Господар му шаље посао, а слуга га извршава, враћа резултат и тражи нови посао. Проблем са овим приступом је што господар може да постане преоптерећен. Овај проблем се делимично може решити *prefetch* механизмом.

Дистрибуирани распоређивач функционише тако што сваки процес одржава своју листу доступних задатака. Процесори са превише доступних задатака их шаљу суседним процесорима. Процесори који остану без задатака потражују посао од суседних процесора.

Критеријуми за успешно извршено мапирање су следећи:

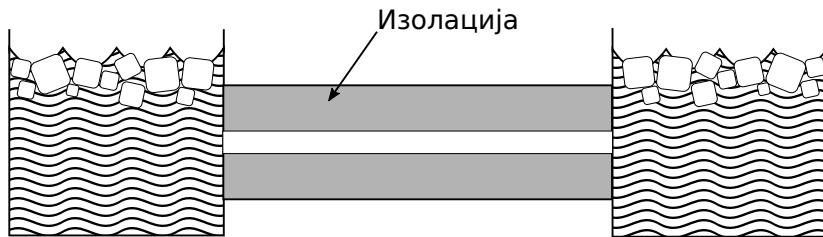
- Разматрани су приступи засновани на једном задатку по процесору, као и вишеструким задацима по процесору.
- Вредновани су и статичка и динамичка алокација задатака.
- Ако је одабрана динамичка алокација, господар не сме да угрожава перформансе.
- Ако је одабрана статичка алокација са вишеструким тасковима, однос између броја задатака и броја процесора је бар 10:1.

Стабло одлучивања приказано је на Слици 3.3. У тексту који следи ће кроз припреме бити објашњена употреба Фостерове методологије на конкретне, нешто поједностављене, проблеме из науке и инжењерства.

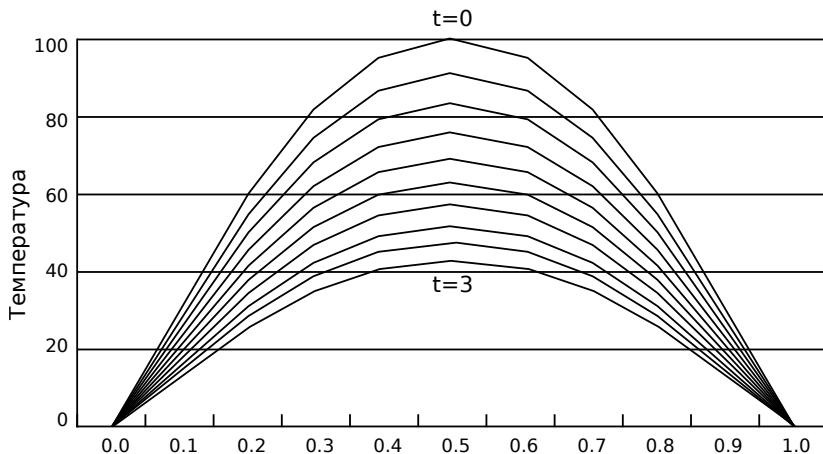
3.4 Пример провођења топлоте

Танак штап од хомогеног материјала је окружен изолацијом, тако да се про- мене температуре у штапу дешавају само као последица размене топлоте ка крајевима штапа и провођења топлоте дуж штапа. Штап је јединичне дужине. Оба краја су изложена мешавини воде и леда температуре 0°C . Почетна температура на растојању x од левог краја штапа је $100 \sin(\pi x)$ (Слике 3.4 и 3.5).

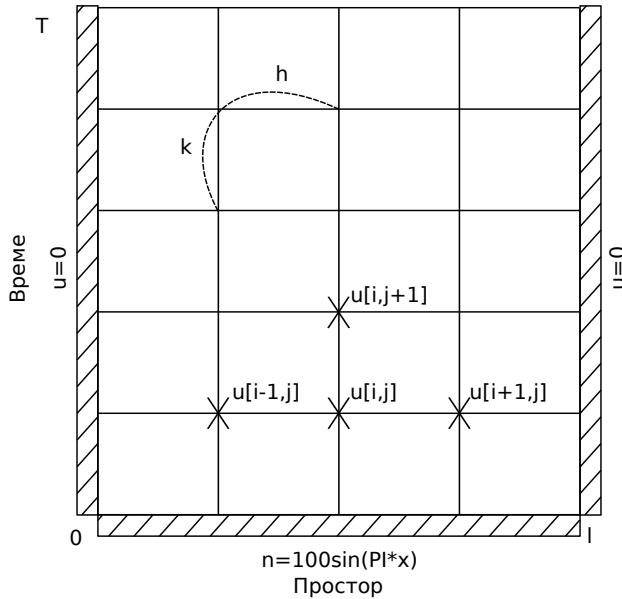
Парцијална диференцијална једначина моделира температуру у било којој тачки штапа у било ком временском тренутку [6]. Ова једначина се решава методом коначних разлика која даје апроксимацију решења за распоред температуре, примењујући просторну и временску дискретизацију. Програмска имплементација решења чува температуру сваке тачке дискретизације у дводимензионој матрици. Сваки ред садржи температурну дистрибуцију штапа у неком тренутку времена. Штап је подељен на n делова дужине h , па стога сваки ред има $n + 1$



Слика 3.4: Експериментална поставка проблема провођења топлоте дуж штапа. На крајевима штапа налази се мешавина воде и леда. Штап је изолован од утицаја спољашње средине.



Слика 3.5: Како време тече, штап се хлади. Метода коначних разлика омогућава израчунавање температуре у фиксном броју тачака у равномерним временским интервалима. Смањење просторног и временског корака доводи до прецизнијег решења.



Слика 3.6: Дискретизација једначине провођења топлоте методом коначних разлика

елемената. Што веће n , мања је грешка у апроксимацији. Време од 0 до T је подељено у m дискретних интервала дужине k , па стога матрица има $m + 1$ редова, Слика 3.6.

Свака тачка $u_{i,j}$ представља елемент матрице који садржи температуру на позицији $i \cdot h$, у тренутку $j \cdot k$. На крајевима штапа је температура увек нула. У почетном тренутку, температура у тачки x је, као што је већ речено, $100 \sin(\pi x)$. Алгоритам иде корак по корак кроз време, користи вредности из тренутка j да би израчунao вредности у тренутку $j + 1$. Формула се овде даје без извођења и гласи:

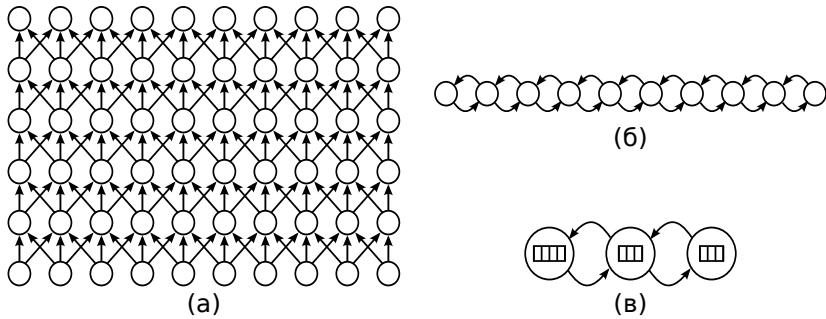
$$u_{i,j+1} = R \cdot u_{i-1,j} + (1 - 2R) \cdot u_{i,j} + R \cdot u_{i+1,j}, \quad (3.1)$$

где је

$$R = \frac{k}{h^2}. \quad (3.2)$$

3.4.1 Партиционисање и комуникација

Као излаз из корака партиционисања, добија се један податак по једној тачки 2Д матрице. Повезујемо по један примитивни задатак са сваком тачком мреже.



Слика 3.7: Партиционисање, комуникација и агломерација код проблема провођења топлоте

Ово је доменска декомпозиција, и то дводимензиона. Сада треба одредити комуникациону шему између примитивних задатака. Као што казује једначина 3.1, сваки унутрашњи примитивни задатак има три излазна канала, као и три улазна канала (Слика 3.7 лево).

3.4.2 Агломерација и мапирање

Задаци који рачунају температуре касније у времену зависе од резултата задатака који рачунају температуру раније у времену. Вертикалне путање канала од дна ка врху означавају да се задаци из исте колоне морају извршавати секвенцијално, тј. један за другим. Зато спајамо задатке у истој колони, тј. оне задатке који су задужени за исту тачку на штапу, без обзира на време, као на Слици 3.7 доле десно.

Добија се линеарни низ, где је сваки члан задужен за рачунање температуре у једној тачки током укупног времена симулације. Комуникација се одиграва искључиво са директним суседима. Број задатака је константан, комуникациона шема регуларна, сваки задатак извршава исти тип прорачуна, па креирамо један задатак по процесору. Сваком процесору се додељује континуални део штапа.

3.4.3 Анализа

Нека χ представља време потребно да се израчуна $u_{i,j+1}$ на основу $u_{i-1,j}$, $u_{i,j}$ и $u_{i+1,j}$. Да би један процесор ажурирао $n - 1$ унутрашњих вредности (две спољне су нула), потребно је $(n - 1) \cdot \chi$ времена. Пошто алгоритам има t временских корака, укупно време извршавања секвенцијалног алгоритма је:

$$T_s = m \cdot (n - 1) \cdot \chi \quad (3.3)$$

Сада треба да проценимо време извршавања паралелног алгоритма. Нека p означава број процесора. Ако су делови штапа једнако подељени по процесорима, време израчунавања за сваку инструкцију је:

$$\chi \cdot \left\lceil \frac{(n-1)}{p} \right\rceil.$$

Заграде $\lceil \rceil$ означавају горњу (неповољнију) вредност, због случаја када број примитивних задатака није дељив бројем процесора.

Међутим, паралелни алгоритам има комуникациони вишак који секвенцијални алгоритам не поседује, па и он мора да се узме у обзир. Нека са λ означена латенција поруке, тј. време потребно процесору да прими, односно пошаље поруку нулте дужине. У нашем моделу, сваки задатак шаље две и прима две поруке у свакој итерацији. Задатак може да шаље само једну поруку у једном тренутку, али зато истовремено са слањем може и да прима поруке, па је комуникациони вишак по итерацији 2λ . Дакле, по итерацији:

$$\chi \cdot \left\lceil \frac{n-1}{p} \right\rceil + 2 \cdot \lambda$$

а укупно:

$$T_p = m \cdot \left(\chi \cdot \left\lceil \frac{n-1}{p} \right\rceil + 2 \cdot \lambda \right). \quad (3.4)$$

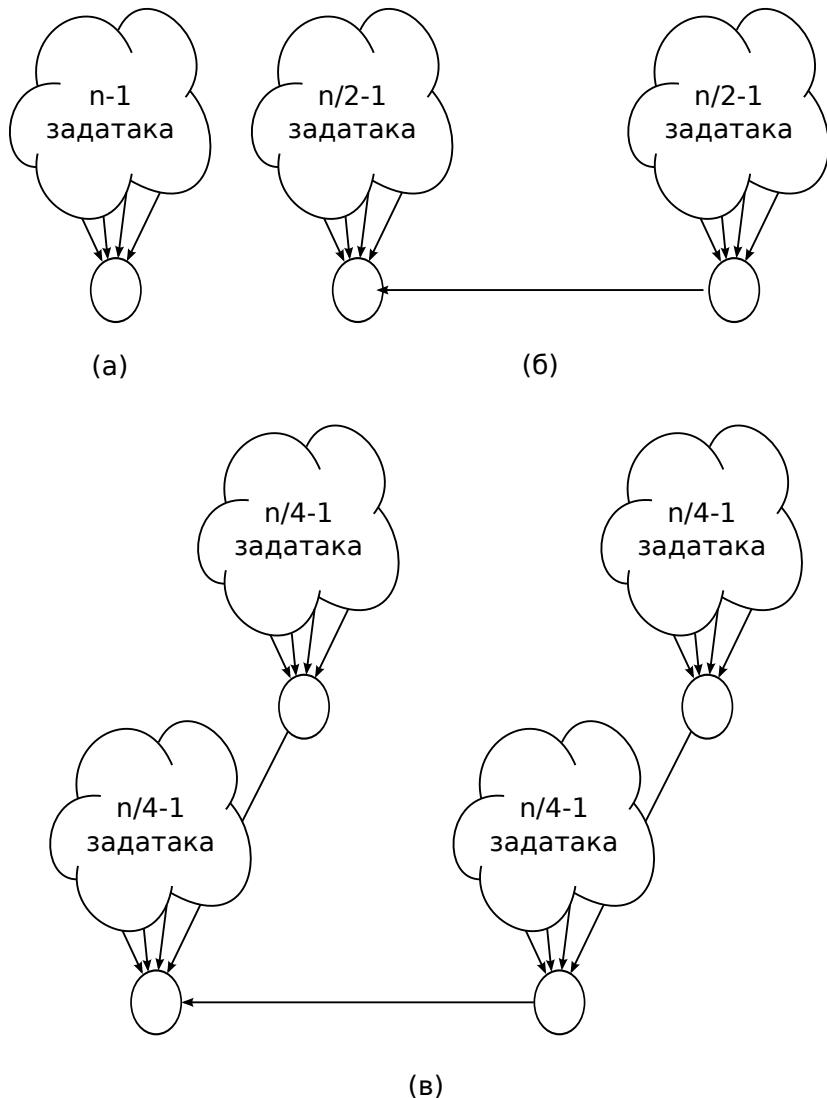
3.5 Максимум низа

Решење претходног проблема коришћењем методе коначних разлика је нумеричка апроксимација. Грешка између израчунатог решења x и тачног решења c је $| \frac{x-c}{c} |$. За овако једноставан проблем, тачно (аналитичко) решење c је познато. Нека сада задатак буде проширење паралелног алгоритма у циљу проналажења максималне грешке.

Ако је дат скуп n вредности $a_0, a_1, a_2, \dots, a_{n-1}$ и асоцијативна бинарна операција \oplus , **редукцијом** се назива операција рачунања $a_0 \oplus a_1 \oplus a_2 \oplus \dots \oplus a_{n-1}$. Примери оваквих операција су сабирање, одузимање, и, или, минимум, максимум, итд. Редукција захтева $n - 1$ операција, па има комплексност $\theta(n)$. Како извршити редукцију на паралелном рачунару?

3.5.1 Партиционисање и комуникација

Низ има n вредности, делимо га на p делова и повежемо по систему један задатак по члану. Циљ је наћи суму свих n вредности. Да бу се израчунала су-ма вредности два задатка, један задатак мора да пошаље своју вредност другом. Идеја је да на крају један задатак (**корен**) има укупан резултат.



Слика 3.8: Приступи решавању проблема максимума низа

Први покушај решавања је да сви задаци шаљу своје вредности корену да их он сам сабере. Време извршавања је:

$$(n - 1) \cdot (\lambda + \chi),$$

што је дефинитивно лошије чак и од секвенцијалног приступа.

Други покушај је са **два квази-корена**, као на Слици 3.8. Две комуникације могу се истовремено одиграти. Такође, две операције сумирања могу симултрано да се обаве. Квази-коренови завршавају посао за:

$$\left(\frac{n}{2} - 1\right) \cdot (\lambda + \chi).$$

Остаје само још да један квази-корен пошаље своју суму другом и да други израчунана коначан резултат, што укупно даје:

$$\frac{n}{2} \cdot (\lambda + \chi).$$

Даљим развојем на четири квази-корена, добија се време:

$$\left(\frac{n}{4} - 1\right) \cdot (\lambda + \chi) + \lambda + \chi + \lambda + \chi = \left(\frac{n}{4} + 1\right) \cdot (\lambda + \chi).$$

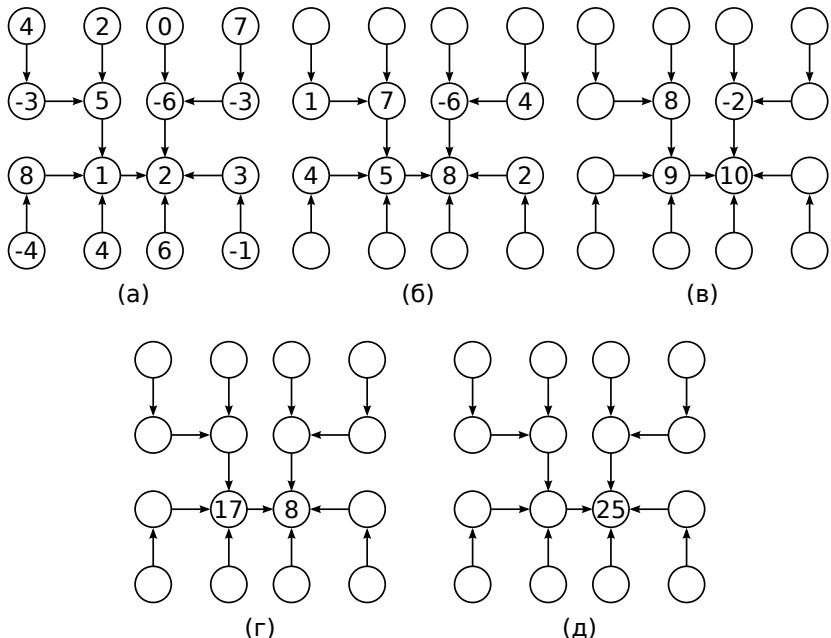
Дакле, што је више квази-коренова, то је алгоритам бржи! Највише можемо да приуштимо $\frac{n}{2}$ квази-коренова. У таквој поставци, у првом кораку половина задатака шаље своје вредности другој половини задатака. Затим задаци примаоци симултрано додају примљене вредности својим сопственим.

Редукција n вредности се врши у $\log_2 n$ комуникационих корака. Комуникацију можемо да представимо као биномно стабло. Ако стабло има 2^k чворова, максимално растојање између корена и било ког чвора је $k = \log_2 n$, тј. дубина ставља. Биномно стабло је најчешћа комуникациона шема у дизајну паралелних алгоритама.

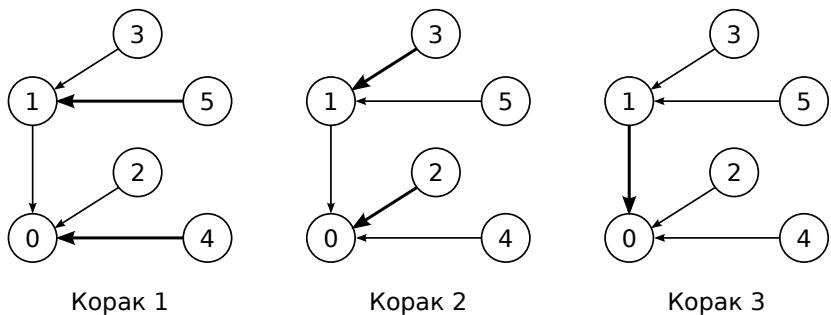
Ако број задатака није степен двојке, додајемо још један корак на почетак алгоритма, који проблем своди на познат (када број задатака јесте степен двојке). Нека је број задатака $n = 2^k + m$, $m < 2^k$. У том новом првом кораку, m задатака шаље вредности, других m таскова прима. m пошиљалаца постаје неактивно, па остаје 2^k активних задатака, чиме сводимо проблем на претходни, Слика 3.10. Дакле, број комуникационих корака у општем случају је

$$\lceil \log_2 n \rceil.$$

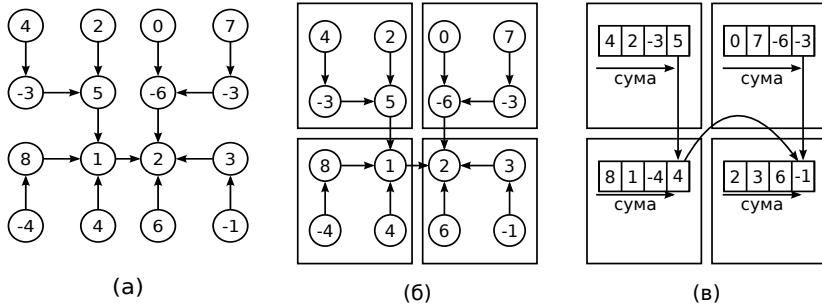
Треба да мапирамо n задатака на p процесора. Претпоставимо да је p такође степен двојке, али је p много мање од n , Слика 3.11. Број задатака је статичан, израчунавања по задатку су тривијална и комуникациона шема је регуларна. Даље, креиратмо p задатака, тако да минимизујемо комуникацију и додељујемо n/p задатака сваком процесору.



Слика 3.9: Решавање проблема максимума низа у логаритамском времену



Слика 3.10: Рачунање максимума у случају да број задатака није степен двојке



Слика 3.11: Агломерација код проблема максимума низа

3.5.2 Агломерација и мапирање

Нема потребе за комуникацијом унутар процесора. Један процесор просто одређује суму својих n/p вредности.

3.5.3 Анализа

Нека је χ време потребно за извршавање бинарне операције, а λ време потребно за трансфер једне целобројне вредности од једног процесора до другог кроз комуникациони канал. Сваки процес поседује највише $\lceil n/p \rceil$ бројева. Процеси се извршавају истовремено и добијају своје подсуме за:

$$\left(\left\lceil \frac{n}{p} \right\rceil - 1 \right) \cdot \chi.$$

У оквиру редукције, извршава се $\lceil \log_2 p \rceil$ корака. Сваки корак укључује једну комуникацију и једно рачунање. Укупно време извршавања паралелног програма може се написати као:

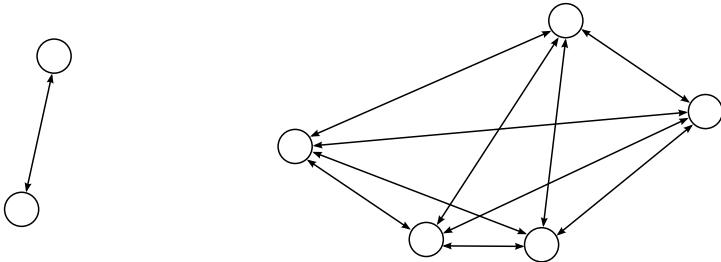
$$T_p = \left(\left\lceil \frac{n}{p} \right\rceil - 1 \right) \cdot \chi + \lceil \log_2 p \rceil \cdot (\lambda + \chi), \quad (3.5)$$

где су:

$\left(\left\lceil \frac{n}{p} \right\rceil - 1 \right) \cdot \chi$ - конкурентно израчунавање сопствене подсуме,

$\lceil \log_2 p \rceil$ - број комуникационих корака потребних за редукцију,

$(\lambda + \chi)$ - сваки корак укључује једну комуникацију и једно рачунање.



Слика 3.12: Поставка проблема n тела. Свако тело делује гравитационом силом на свако друго тело.

3.6 Проблем n тела

Ради се о моделирању кретања n тела са различитом масом, која делују једна на друге гравитационим силама, у две димензије. Слично би се решавао проблем интеракције тела при деловању било које друге централне сile, нпр. електростатичке. Гравитациона сила зависи од масе и растојања, а дomet јој је бесконачан. Стога се израчунавања врше над **свим паровима тела** (Слика 3.12), па је стога комплексност секвенцијалног алгоритма $\theta(n^2)$ по итерацији. У току сваке итерације рачунамо нову позицију и вектор брзине за сваку честицу, на основу положаја свих осталих честица.

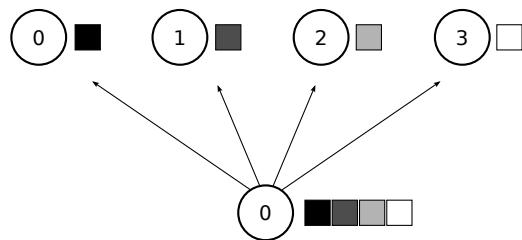
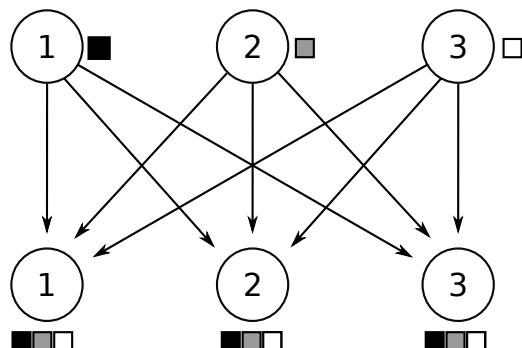
Систем се дефинише својим иницијалним стањем, које чине сви вектори положаја сваког од n тела, као и њихове брзине $(x_i, y_i, v_{xi}, v_{yi}), \forall i \in \{1..n\}$.

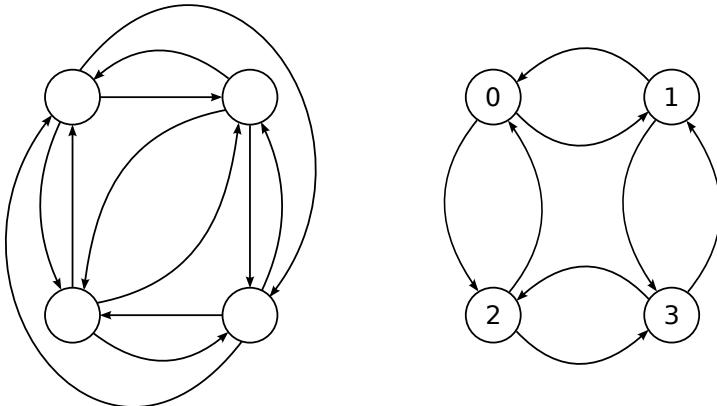
3.6.1 Партиционисање и комуникација

Користи се домен-декомпозиција. Претпоставимо да имамо један примитивни задатак по телу. Задатак од података поседује позицију тела и вектор брзине. У свакој итерацији, задатак узима позиције свих осталих честица и рачуна укупну силу, а из ње нову позицију и вектор брзине. За комуникацију је погодно користити операцију *gather*. *Gather* је глобална комуникација, која скуп података дистрибуиран међу групом задатака све сакупља на једном месту. За разлику од редукције, не добија се један једини резултат из свих података, већ се подаци спајају у колекцију, Слика 3.13.

All-gather операција је слична *gather*, осим што на крају комуникације сваки таск има копију целог скупа података, Слика 3.14.

За решавање проблема n тела погодна је управо операција *All-gather*, јер свако тело треба да поседује податке о позицији свих осталих тела, како би се исправно

Слика 3.13: *Gather* операцијаСлика 3.14: *Allgather* операција



Слика 3.15: Лево: Комуникација типа комплетан граф. Десно: Комуникација типа хиперкоцка

израчунала укупна сила. Да иначки гледамо, начин да остваримо ову комуникацију је да ставимо канал између сваког пара задатака.

Током сваког комуникационог корака, сваки задатак шаље свој вектор другом задатку. Након $n - 1$ корака, сваки задатак поседује позиције свих осталих честица, Слика 3.15 лево.

3.6.2 Може ли ефикасније?

Ако имамо две честице, сваки задатак шаље координате тела преко једног канала и прима вредност преко другог канала. Ако имамо четири честице, након једног корака задаци означени са 0 и 1 имају вредности положаја ν_0 и ν_1 , док задаци 2 и 3 оба имају вредности ν_2 и ν_3 . Ако задатак 0 размени свој пар честица са задатком 2, а задатак 1 замени свој пар са задатком 3, сви ће имати по четири честице, као на Слици 3.15 десно!

Дакле, потребно је само $\log_2 n$ корака да сваки задатак добије позиције свих осталих тела. У првом кораку, поруке имају дужину 1, а у другом дужину 2. У i -том кораку, поруке имају дужину 2^{i-1} . Оваква комуникација функционише на принципу хиперкоцке и често се користи у *all-to-all* разменама података.

Агломерација и мапирање

Претпоставка модела је да има много више честица n него процесора p . Претпоставимо да је n дељиво са p . Дакле, мапирајмо један задатак по процесору, као и у претходном примеру, а онда агломеризујемо n/p тела за сваки процес. Сада агломеризована *all-gather* операција захтева $\log_2 p$ корака. У првом кораку поруке

имају дужину n/p , у другом $2n/p$, итд.

3.6.3 Анализа

Новина о којој треба повести рачуна је чињеница да дужина порука варира, што имплицира да време трансфера поруке зависи од њене дужине. Ако са λ означимо време потребно да се иницира порука (латенција), а са β количину података која може да се пошаље кроз канал у јединици времена (проток), онда слање поруке дужине n траје:

$$\lambda + \frac{n}{\beta}.$$

Дакле, што је проток већи, дужина трајања трансфера је краћа. За први начин комуникације (комплетан граф), време комуникације износи:

$$(p-1) \cdot \left(\lambda + \frac{\frac{n}{p}}{\beta} \right) = (p-1) \cdot \lambda + \frac{n \cdot (p-1)}{\beta \cdot p}.$$

За други начин комуникације (хиперкоцка), време комуникације износи:

$$\sum_{i=1}^{\log_2 p} \left(\lambda + \frac{2^{i+1} \cdot n}{\beta p} \right).$$

Узевши у обзир да вредности $2^0, 2^1, \dots, 2^{\log_2 p - 1}$ чине геометријски низ са параметрима $a_1 = 1$, $q = 2$ и дужине $n = \log_2 p$, суме овог низа се може добити следећим формулама:

$$S_n = a_1 \cdot \frac{q^n - 1}{q - 1},$$

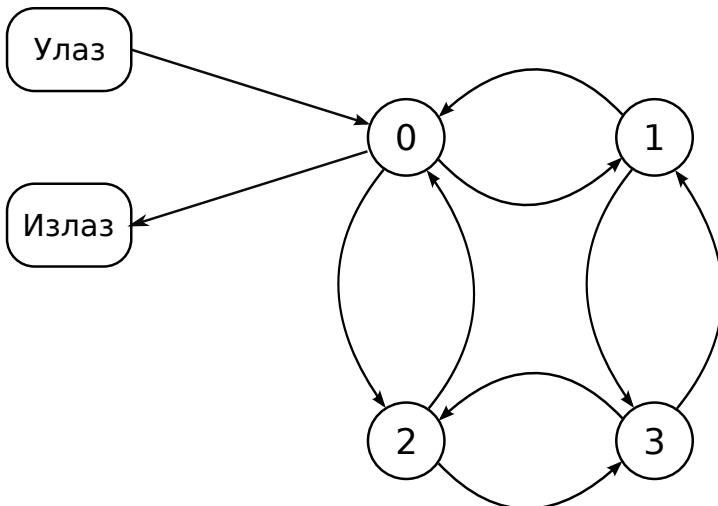
$$S_{\log_2 p} = 1 \cdot \frac{2^{\log_2 p} - 1}{2 - 1} = p - 1.$$

На основу тога, може се добити и укупно време комуникације као:

$$\sum_{i=1}^{\log_2 p} \left(\lambda + \frac{2^{i+1} \cdot n}{\beta p} \right) = \lambda \cdot \log_2 p + \frac{n(p-1)}{\beta p}. \quad (3.6)$$

Сваки процес је задужен за оквирно $\frac{n}{p}$ елемената. Рецимо да је за једно израчунавање силе, нове брзине и положаја потребно χ времена. Сви процеси овај прорачун врше истовремено, па је време израчунавања $\chi \lceil n/p \rceil$, за један временски корак. Очекивано време израчунавања паралелног алгоритма по једном временском кораку је:

$$\lambda \cdot \log_2 p + \frac{n(p-1)}{\beta p} + \chi \left\lceil \frac{n}{p} \right\rceil.$$



Слика 3.16: Додавање улаза/излаза

3.6.4 Додавање уноса података

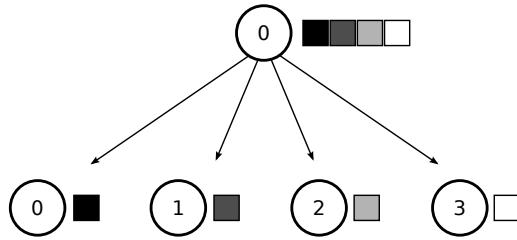
Програм који имплементира алгоритам моделирања кретања n тела не би могао да функционише без додатних механизама за улаз почетног стања система и излаз коначног стања система путем У/И канала. На почетку треба унети иницијалне векторе позиција и брзина за свих n тела, а на крају исте те величине уписати у фајл. Комерцијални паралелни рачунари често имају специјалне У/И системе, док се мањи кластери углавном ослањају на екстерне фајл сервере који чувају обичне UNIX фајлове, када је један једини процес одговоран за извођење У/И операција над фајлом. У овом случају, нећемо додати специјални процес за У/И, већ тај посао додељујемо процесу са рангом 0, Слика 3.16. О напреднијој употреби MPI-2 стандарда у сврху имплементације паралелног улаза и излаза може се прочитати у Глави II ове књиге.

У/И процес отвара фајл и чита позиције и брзине свих n тела. И вектори позиција и вектори брзина имају по две координате, јер је проблем, као што је већ речено, дводимензиони. Ако је $\lambda_{io} + \frac{4n}{\beta_{io}}$ време потребно за унос или испис n елемената, онда за читање позиције и брзине треба:

$$\lambda_{io} + \frac{4n}{\beta_{io}}.$$

Комуникација

Пошто је У/И задатак учитао честице, сада мора их раздели, тако да сваки процес добије приближно $\frac{n}{p}$ елемената. Ова глобална комуникациона операција

Слика 3.17: Операција *scatter*

назива се ***scatter*** и представља супротност операцији *gather*.

Први начин: У/И процес шаље тачно $\frac{n}{p}$ честица сваком од преосталих процеса. Другим речима, шаље $p - 1$ поруку, где је свака порука дужине $4 \cdot \frac{n}{p}$. Време потребно за извођење овог приступа је:

$$(p - 1) \cdot \left(\lambda + \frac{4n}{\beta p} \right)$$

Очигледно је да овај начин није најефикаснији, јер комуникација међу процесима није балансирана.

Други начин: У првом кораку, У/И процес шаље половину података другом процесу. У другом кораку сваки процес са половином шаље четвртину података претходно неактивним процесима. Сада четири процеса имају по четвртину података. У трећем кораку, четири процеса са четвртином података шаљу по осмину претходно неактивним процесима, итд. На овај начин се операција *scatter* изврши за $\log_2 p$ корака, као на Слици 3.17.

Време потребно за улаз и излаз је:

$$2 \cdot \left(\lambda_{io} + \frac{4n}{\beta_{io}} \right),$$

док је време потребно за расподелу почетних стања процесима:

$$\sum_{i=1}^{\log_2 p} \left(\lambda + \frac{4n}{2^i \cdot \beta p} \right). \quad (3.7)$$

Сума одређена једначином (3.7) може се израчунати ако се зна суме геометријског низа:

$$S_n = \sum_{i=1}^{\log_2 p} \frac{1}{2^i},$$

где је $n = \log_2 p$. Елементи овог низа су редом:

$$\frac{1}{2}, \frac{1}{2^2} = \frac{1}{4}, \frac{1}{2^3} = \frac{1}{8}, \dots, \frac{1}{2^{\log_2 p}} = \frac{1}{p}.$$

Како су параметри овог геометријског низа $a_1 = \frac{1}{2^0}$ и $q = \frac{1}{2}$, можемо да пишемо:

$$S_{\log_2 p} = \frac{1}{2} \cdot \frac{\left(\frac{1}{2}\right)^{\log_2 p} - 1}{\frac{1}{2} - 1} = \frac{1}{2} \cdot \frac{\frac{1}{2^{\log_2 p}} - 1}{-\frac{1}{2}} = -\left(\frac{1}{p} - 1\right) = \frac{p-1}{p}$$

На основу тога, време потребно за расподелу почетног стања процесима износи:

$$\sum_{i=1}^{\log_2 p} \lambda + \frac{4n}{2^i \cdot \beta p} = \lambda \cdot \log_2 p + \frac{4n(p-1)}{\beta p}. \quad (3.8)$$

Уколико би се пак за расподелу искористио први начин, добили бисмо:

$$(p-1) \cdot \lambda + \frac{4n(p-1)}{\beta p}. \quad (3.9)$$

Овде треба приметити да је време преноса података $\frac{4n(p-1)}{\beta p}$ идентично и за први и за други начин. У првом начину, сваки податак се шаље само једном, али без конкурентности. У другом (ефикаснијем) начину, свака порука се шаље $\log_2 p$ пута, али има конкурентности, осим у случају да се комуницира разводником (*hub*) или неким другим дељеним медијумом.

Анализа

Ако се у обзир узму сви елементи потребни за имплементацију проблема n тела у m временских корака, укупно време рачунања може да се изрази следећом једначином:

$$T_p = 2 \left(\lambda_{io} + \frac{4n}{\beta_{io}} \right) + 2 \left(\lambda \log_2 p + \frac{4n(p-1)}{\beta p} \right) + m \left(\lambda \log_2 p + \frac{2n(p-1)}{\beta p} + \chi \left\lceil \frac{n}{p} \right\rceil \right) \quad (3.10)$$

Расподела времена паралелног извршења по компонентама је следећа:

- $2 \left(\lambda_{io} + \frac{4n}{\beta_{io}} \right)$ – улаз и излаз,
- $2 \left(\lambda \log_2 p + \frac{4n(p-1)}{\beta p} \right)$ – *scatter* на почетку и *gather* на крају симулације,
- $\lambda \log_2 p + \frac{2n(p-1)}{\beta p}$ – *all-gather* позиција у сваком кораку,
- $\chi \left[\frac{n}{p} \right]$ – израчунавања у сваком кораку.

Глава 4

Програмирање разменом порука

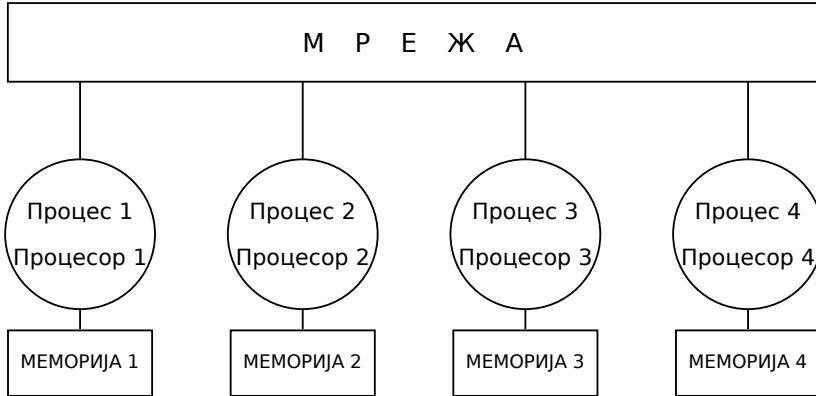
4.1 Историјат, концепт и основне MPI функције

Како стоје ствари, ни један наменски паралелни језик високог нивоа за сада није општеприхваћен. Уместо тога се постојећим секвенцијалним језицима дођају библиотечке функције које обављају размену порука између процеса. MPI (*Message Passing Interface*) данас представља *de facto* стандард за размену порука која подржава паралелно програмирање. Сваки паралелни рачунар подржава MPI стандард, а доступне су и бесплатне библиотеке које овај стандард делимично или потпуно имплементирају.

4.1.1 Историјат

Кратак историјат библиотека паралелних функција дат је следећим ставкама:

- Касних 60-их су много компаније почеле да производе мултикомпјутере. Тадашње окружење се састојало од обичног секвенцијалног језика (С или FORTRAN) и библиотеке за размену порука. Сваки произвођач је креирао своје сопствене библиотеке порука, па стога није могло бити портабилности апликација. Програмери, а потом и компаније су почели да се залажу за увођење стандарда у ову област.
- 1989. године је развијена библиотека PVM (*Parallel Virtual Machine*) у *Oak Ridge* националној лабораторији [10]. PVM је подржавао извршавање паралелних програма на хетерогеном скупу серијских и паралелних рачунара. Користио се унутар поменуте лабораторије, а тек је верзија 3 пуштена у јавност 1993. и одмах је постала популарна.
- 1992. године је Центар за развој паралелног рачунарства организовао радионицу у вези са увођењем стандарда у размену порука међу процесима. Та да је и започет рад на MPI стандарду, новом стандарду који ће објединити



Слика 4.1: Модел размене порука

најефикасније постојеће библиотеке које су независно развијали производи супер-рачунара.

- 1994. Објављена је прва верзија MPI стандарда [7].
- 1997. Објављена је друга верзија MPI стандарда која укључује функционалности паралелног улаза и излаза [8].
- Данас је MPI доминантан стандард у паралелном рачунарству. Верзија MPI стандарда која је тренутно актуелна је 3.1 [9].

4.1.2 Модел размене порука

Модел размене порука се директно пресликова у модел задатак/канал, који је обрађен у претходном поглављу.

Хардвер чини скуп процесора, сваки са својом локалном меморијом. Процесори имају приступ само инструкцијама и подацима у својој локалној меморији. Међутим, комуникациона мрежа подржава размену порука дајући индиректан приступ туђој локалној меморији, Слика 4.1.

Задатак из задатак/канал модела постаје процес у моделу размене порука. Интерконекциона мрежа значи да постоји канал између свака два процеса, тј. да може да комуницира свако са сваким. Међутим, и даље користимо стратегије за смањивање комуникационог вишка. Корисник специфицира број комуникационих процеса на почетку извршавања програма. Овај број остаје константан током целокупног времена извршења. Сви процеси извршавају исти програм, али пошто сваки има јединствени идентификациони број, различити процеси могу извршавати различите делове програма. Процес извршава израчунавања

над својим локалним промиљивама и комуницира са другим процесима и У/И уређајима.

4.1.3 Предности модела размене порука

Програми су преносиви на разне MIMD архитектуре. MPI је природно решење за мултикомпјутере, али ради сасвим добро и на мултипроцесорима, где дељене промењиве користи као бафере за поруке. Поред тога, прави разлику између брже меморије са директним приступом и удаљене меморије са индиректним приступом. Омогућава максимизовање локалних израчунавања и минимизовање комуникација. Даје програмима могућност да управљају хијерархијом меморије.

Поруке служе и за комуникацију и за синхронизацију. Процес не може да прими поруку док је други процес не пошаље. Зато чак и празна порука може да има значење.

4.1.4 Основне MPI функције

- `MPI_Init (&argc, &argv)` - прва MPI функција коју позива сваки процес. Омогућава даље позивање функција MPI библиотеке. Налаже систему да уради сва потребна подешавања паралелног окружења. Не мора да буде прва команда у програму уопште, већ само прва MPI команда.
- Комуникатор је затворени објекат који пружа окружење за размену порука међу процесима. `MPI_COMM_WORLD` је продразумевани комуникатор, укључује све процесе и углавном је довољан, али је могуће користити и нове комуникаторе, уколико су нам потребне независне комуникационе групе. Процеси унутар комуникатора су уређени, сваки има свој јединствени ранг (идентификациони број). Ако има p процеса, рангови иду од 0 до $p - 1$. Процес може да искористи свој ранг да одреди за који део израчунавања и података је задужен.
- `MPI_Comm_size (MPI_COMM_WORLD, &p)` - одређује укупан број процеса у комуникатору. Први аргумент је комуникатор, док се број процеса враћа кроз други аргумент.
- `MPI_Comm_rank (MPI_COMM_WORLD, &id)` - одређује ранг процеса који је позвао ову функцију унутар прослеђеног комуникатора. Први аргумент је комуникатор. Ранг процеса (у опсегу од 0 до $p - 1$) се враћа кроз други аргумент.
- `MPI_Finalize ()` - позива се пошто процес заврши све позиве MPI библиотеке, дозвољава систему да ослободи ресурсе (као што је меморија) које је MPI процес алоцирао.

- Колективна комуникација је комуникациона операција у којој група процеса ради заједно у циљу дистрибуирања или скупљања једне или више вредности. Пример је редукција.
- MPI_Reduce врши једну или више редукција над вредностима послатим од стране свих процеса у комуникатору.

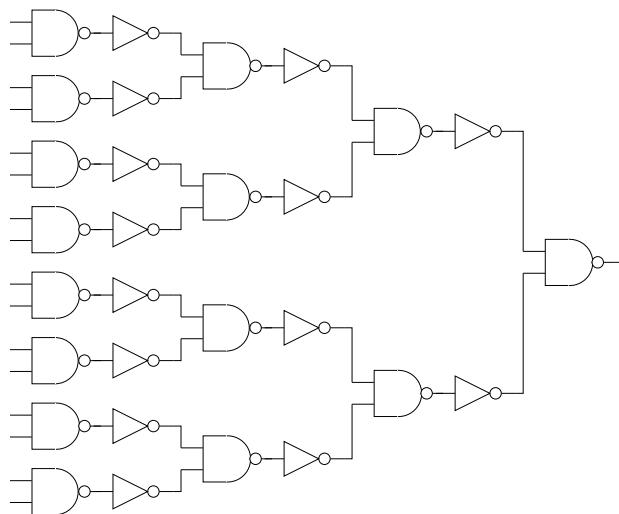
```
int MPI_Reduce(void *operand, void *result, int count,
               MPI_Datatype type, MPI_Op operator,
               int root, MPI_Comm comm)
```

- count означава колико се редукција врши. Сваки процес шаље count вредности и свака од ових вредности учествује у посебној редукцији.
- operand је локација елемената за прву редукцију. Ако је count веће од 1, онда листа елемената за све редукције заузима континуални блок меморије.
- type је тип елемената који се редукују (MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR).
- operator одређује врсту редукције. Редукциони оператори су MPI_SUM, MPI_PROD, MPI_MIN, MPI_MAX, MPI_LAND, MPI_LOR, MPI_LXOR, MPI_BAND, MPI_BOR, MPI_BXOR, ...
- root је ранг процеса који ће имати резултате.
- result је локација првог редукционог резултата. Има значење само за root процес.
- comm је име комуникатора тј. скупа процеса који учествују у редукцији.

4.2 Алгоритам одређивања функције истине логичког кола

Дато је логичко коло са 16 улаза и једним излазом, приказано на Слици 4.2. Треба одредити за које комбинације улазних вредности ће коло дати излаз 1. Овај проблем је важан за дизајн и верификацију логичких кола. Проблем је NP комплексан, тј. нема познатих алгоритама који раде у полиномијалној комплексности. Решење је реалтивно једноставно: методом исцрпљивања. Дакле, треба испробати све могуће бинарне комбинације. Имамо 16 улаза и сваки може да има две вредности, 0 и 1, па је укупно $2^{16} = 65536$ улазних комбинација.

Партиционисање се своди на функционалну декомпозицију, тј. повезујемо по један задатак са сваком комбинацијом улаза. Овакви задаци су потпуно независни, па могу да се извршавају паралелно. Ако се посматра задатак/канал модел



Слика 4.2: Одређивање функције истине логичког кола

овог проблема, очигледно је да нема канала између њих. Овај тип проблема се назива и *embarrassingly parallel problem*.

4.2.1 Агломерација и мапирање

Дакле, имамо фиксни број задатака, нема комуникације међу њима, а време потребно за сваки задатак варира, зато што за неке битовске комбинације одмах може да се види да задовољавају (дају нулу), а за неке треба више времена. Користећи стабло одлучивања, бирамо стратегију цикличног мапирања задатака на процесоре.

Креирамо по један процес по процесору. Са n делова посла и p процесора додељујемо процесору сваки p -ти део посла. На пример, за случај 5 процесора и 12 делова посла имамо:

$$\begin{aligned} P_0 &: 0, 5, 10 \\ P_1 &: 1, 6, 11 \\ P_2 &: 2, 7 \\ P_3 &: 3, 8 \\ P_4 &: 4, 9 \end{aligned}$$

Посао број k се додељује процесу $k \bmod p$. Сваки процес ће да провери све своје комбинације и штампаће оне које задовољавају.

Листинг 4.1: Одређивање функције истине логичког кола

```

1 #include "mpi.h"
2 #include <stdio.h>
3
4 int main (int argc, char *argv[]) {
5     int i;
6     int id;           /* Rang procesa */
7     int p;            /* Broj procesa */
8     void check_circuit (int, int); /* Funkcija kola */
9
10    MPI_Init (&argc, &argv);
11    MPI_Comm_rank (MPI_COMM_WORLD, &id);
12    MPI_Comm_size (MPI_COMM_WORLD, &p);
13
14    for (i = id; i < 65536; i += p)
15        check_circuit (id, i);
16
17    printf ("Proces %d je završio\n", id);
18    fflush (stdout);
19    MPI_Finalize();
20    return 0;
21 }
```

Сада желимо да додамо функционалност која ће омогућити рачунање укупног броја решења. Сваки процес ће у некој својој променљивој да чува број решења која је пронашао, а на самом крају ће сарађивати у циљу израчунавања глобалне суме свих вредности. За овај задатак је погодно користити редукцију. Додајемо две нове променљиве у `main()`, као на Листингу 4.2.

Листинг 4.2: Редукција

```

1 int solutions; // broj resenja koja je pronasao jedan proces
2 int global_solutions // ukupan broj resenja svih procesa, koristi je
3 // samo procces sa rangom 0
4 int check_circuit(int, int) // funkcija kola vraca 1 ako zadovoljava
5 // , a 0 ako ne zadovoljava
6 solutions = 0;
7 for(i=id;i<65536;i+=p)
8     solutions += check_circuit(id, i);
9 MPI_Reduce(&solutions, &global_solutions, 1, MPI_INT, MPI_SUM, 0,
10 MPI_COMM_WORLD);
```

4.3 Процедура покретања MPI програма

4.3.1 Компајлирање

Ако смо сачували програм као `sat1.c`, позивамо:

```
mpicc -o sat1 sat1.c
```

што је, у ствари, краћи начин писања за:

```
gcc -o sat1 sat1.c -lmpi
```

Дакле скрипта `mpicc` компајлира и линкује MPI програме написане у програмском језику С. Добијени извршни фајл је `sat1`.

4.3.2 Покретање

Покретање се врши командом `mpirun`. Заставица `-np` означава број процеса који треба креирати.

```
mpirun -np 16 sat1
```

Редослед у ком се појављује испис само делимично одсликава редослед стварних догађаја исписа у паралелном рачунару. Ако процес А штампа две поруке, прва ће бити штампана пре друге. Али, ако процес А штампа поруку, па процес Б штампа поруку, то не значи да ће се порука процеса А појавити пре поруке процеса Б, већ редослед зависи од сплета догађаја у систему.

4.3.3 Мерење перформанси

Стандардном UNIX командом `time` можемо да меримо време од почетка до краја извршења програма. Међутим, да бисмо добили прецизнију слику о перформансама, боље је не узимати у обзир иницијализацију MPI процеса и У/И операције, већ само паралелизовани део кода између учитавања и исписа. За намену мерења времена, MPI стандард прописује следеће функције:

```
double MPI_Wtime(void)
double MPI_Wtick(void)
```

Функција `MPI_Wtime()` враћа број секунди протеклих од неког тренутка у прошлости, а функција `MPI_Wtick()` враћа прецизност резултата који враћа `MPI_Wtime()`.

Меримо перформансе тако што стављамо позиве функције `MPI_Wtime` пре и после жељене секције кода. Разлика између ова два резултата је време извршавања секције. Међутим, MPI процеси који се извршавају на различитим процесорима могу почети своје извршавање у различито време, па ће сваки процес израчунати значајно другачује време извршења.

Овај проблем решавамо баријерном синхронизацијом пре првог позива `MPI_Wtime`. Ни један процес не може прећи баријеру док сви процеси не стигну до ње. Баријера осигуруја да ће сви процеси ући у посматрану секцију кода у исто време:

```
int MPI_BARRIER (MPI_Comm comm)
```

Једини аргумент је комуникатор који учествује у баријери. На пример:

```
double elapsed_time;
MPI_Init(&argc, &argv);
MPI_BARRIER(MPI_COMM_WORLD);
elapsed_time = -MPI_Wtime();
. . .
MPI_Reduce();
elapsed_time += MPI_Wtime();
```

Глава 5

Ератостеново сито

5.1 Секвенцијални алгоритам и паралелизабилност

Ератостеново сито (решето) је једноставан алгоритам за добивање свих простих бројева мањих од оног изабраног (n), Слика 5.1. Осмислио га је грчки математичар, географ и астроном Ератостен. Може се дефинисати следом корака:

1. Креирати листу неозначених природних бројева $2, 3, \dots, n$
2. $k = 2$
3. Понављај
 - (а) Означи све бројеве дељиве са k између k^2 и n .
 - (б) Следеће k је најмањи неозначен број већи од k .
све док не буде задовољен услов $k^2 > n$.
4. Неозначени бројеви су прости.

Ератостеново сито није практично ако је намена потрага за великим прстим бројевима са стотинама цифара, из разлога што алгоритам има комплексност $\theta(n \ln \ln n)$. У програмском језику С за означавање користимо низ од $n - 1$ char-ова (са индексима $0, 1, 2, \dots, n - 2$) да бисмо представили природне бројеве $2, 3, \dots, n$. Boolean вредност на индексу i означава да ли је број $i + 2$ означен или не.

X	(2)	(3)	X	(5)	X	(7)	X	-9	X
(11)	X	(13)	X	-15	X	(17)	X	(19)	X
-21	X	(23)	X	25	X	-27	X	(29)	X
(31)	X	-33	X	35	X	(37)	X	-39	X
(41)	X	(43)	X	-45	X	(47)	X	-49	X
-51	X	(53)	X	55	X	-57	X	(59)	X
(61)	X	-63	X	65	X	(67)	X	-69	X
(71)	X	(73)	X	-75	X	-77	X	(79)	X
-81	X	(83)	X	85	X	-87	X	(89)	X
-91	X	-93	X	95	X	(97)	X	-99	X

Слика 5.1: Ератостеново сито

5.1.1 Извори паралелизма

Главни део алгоритма је означавање елемената низа. Зато користимо доменску декомпозицију, тј. делимо низ на $n - 1$ елемената и повезујемо по један примитивни задатак са сваким од ових елемената. Кључно паралелно израчунавање је садржано у кораку 3(a):

```

for all  $j$  where  $k^2 \leq j \leq n$  do
    if  $j \bmod k = 0$  then
        mark  $j$  (it is not a prime)
    end if
end for

```

Сваки задатак проверава за своје j . Онда иде 3(б), где су нам потребне две комуникације, и то редукција да се одреди нова вредност k и емисија да се ново k пошаље свим задацима. Добро је што има пуно паралелизма, лоше је што има много редукција и емисија (по једна у свакој итерацији). Следеће питање којим ћемо се бавити је како агломеризовати примитивне задатке, како балансирати

израчунавања међу процесима и како смањити комуникацију.

5.2 Циклична или блок декомпозиција?

После агломерације, један процес ће бити одговоран за групу елемената низа. Ово груписање података се назива декомпозицијом података.

5.2.1 Циклична декомпозиција података

Процес 0 је одговоран за бројеве $2, 2+p, 2+2p, \dots$, док је процес 1 је одговоран за бројеве $3, 3+p, 3+2p, \dots$. Предност овакве декомпозиције је да је за дати индекс i лако одредити који процес је задужен за тај индекс (процес $i \bmod p$). Мана је лоше балансирање оптерећења. Нпр. ако имамо 2 процеса и означавамо умношке броја 2, процес 0 ће означити $\lceil (n-1)/2 \rceil$ елемената, док процес 1 неће означити ни један елемент низа. Друга мана се садржи у чињеници да нисмо смањили ни редукције ни емисије.

5.2.2 Блок декомпозиција података

У случају блок декомпозиције, делимо низ на p непрекидних блокова, отприлике једнаке дужине. Овако се доље балансира оптерећење, али је нешто теже одредити припадност члана низа процесу у случају да n није дељиво са p . Како балансирати оптерећење ако n није дељиво са p ?

Нпр. $n = 1024, p = 10, 1024/10 = 102$. Ако сваком дамо 102, остаје 4 вишке. Не можемо да дамо сваком 103, а последњем колико остане, јер можда не остане ништа, а то би направило проблеме у балансу оптерећења. Дакле, сваки процес треба да добије или $\lceil n/p \rceil$ или $\lfloor n/p \rfloor$ елемената. Размотрићемо две методе.

Прва метода

Рачунамо $r = n \bmod p$. Ако је $r = 0$, сви блокови имају величину n/p . Ако је $r > 0$, првих r процеса има блок величине $\lceil n/p \rceil$, а осталих $p-r$ процеса добијају блок величине $\lfloor n/p \rfloor$. На пример: $\lfloor 17/7 \rfloor = 2, \lceil 17/7 \rceil = 3, 17 \bmod 7 = 3$.

Први елемент који контролише процес i је: $i\lfloor n/p \rfloor + \min(i, r)$ – сваки процес добије $\lfloor n/p \rfloor$ елемената и можда још један елемент. Процес 0 ($i = 0$) креће од нултог елемента и има сигурно $\lfloor n/p \rfloor$, а ако је $r > 0$, онда има још један. Процес 1 ($i = 1$) прескаче првих $\lfloor n/p \rfloor$ елемената ($i\lfloor n/p \rfloor$) и можда још један. Процес 2 прескаче $2\lfloor n/p \rfloor$ елемената, онај један додатни за нулти, и онај један додатни за први, укупно $2(i)$. Када r постане мање од i , процеси више не добијају тај један додатни елемент, али зато сваки прескаче r претходних додатних.

Последњи елемент који контролише процес i је: $(i+1)\lfloor n/p \rfloor + \min(i+1, r) - 1$. Дакле, идеја је да се израчуна који је први елемент следећег процеса и одузме се 1.

Процес који контролише елемент j се добија као $\min(\lfloor j/(\lfloor n/p \rfloor + 1) \rfloor, \lfloor (j - r)/\lfloor n/p \rfloor \rfloor)$ – Први израз дели j са $\lceil n/p \rceil$, осим кад је n дељиво са p . Ово важи за првих r процеса који имају $\lceil n/p \rceil$ елемената. После тога се од j одузима r (они додатни елементи) па то дели $\lceil n/p \rceil$ (толико сваки процес има без додатних елемената). Ово важи за последњих $p - r$ елемената.

Сви ови изрази су прилично компликовани. Први и последњи елемент могу да се израчунају одмах на почетку, али припадност не може, па се повећава комплексност програма.

Друга метода

Друга метода не концентрише све веће блокове на првим процесима. Први елемент који контролише процес i је in/p – Процеси добијају $\lceil n/p \rceil$ елемената, док остатак при дељењу не пређе p , а кад пређе, процес добија $\lceil n/p \rceil$ и тако у круг. Последњи елемент који контролише процес i : $\lfloor (i+1)n/p \rfloor - 1$ – Израчунамо први елемент следећег процеса и одузмемо 1. Процес који контролише елемент j је $\lfloor (p(j+1) - 1)/n \rfloor$.

Други приступ је ефикаснији, зато што за сваки од ова три обрасца захтева мањи број операција, поготову зато што дељење у програмском језику С аутоматски ради `floor`.

Глобални и локални индекси

Дефинишими сада макрое за блок декомпозицију који се базирају на другом методу блок декомпозиције:

```
#define BLOCK_LOW (id,p,n) ((id)* (n) / (p))
#define BLOCK_HIGH (id,p,n) (BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE (id,p,n) (BLOCK_HIGH(id,p,n) -
                           BLOCK_LOW(id,p,n)+1)
#define BLOCK_OWNER (index,p,n) (((p*((index)+1))-1) / (n))
```

Сада треба да уведемо појмове глобалног и локалног индекса. Ознака за локални индекс је i , а за глобални индекс gi . Ево како се њима може баратати програмски:

```
size=BLOCK_SIZE...
for (i=0; i<SIZE; i+t) {
    gi = i + BLOCK_LOW;
    ...
}
```

Највећи прост број који користимо за просејавање је \sqrt{n} , а први процес има $\lfloor n/p \rfloor$ елемената. Ако први процес (онај са рангом 0) садржи све бројеве до \sqrt{n} ,

свако k које просејава се налази на првом процесу, па нам корак редукције није ни потребан, већ само процес са рангом 0 емитује k осталима у свакој итерацији. Очекујемо да n буде макар у милионима, па сигурно важи да је $\lfloor n/p \rfloor > \sqrt{n}$.

Још једна мала модификација је да не морамо проверавамо стално да ли је елемент низа дељив са k , што би захтевало $n/p \bmod$ операција. Једноставније је да нађемо само први умножак k у блоку (зовимо га j), и онда маркирамо бројеве $j, j+k, j+2k, \dots$. За ово нам треба $(n/p)/k$ операција додељивања, што је значајно ефикасније.

5.2.3 Паралелни алгоритам

Принцип рада паралелног алгоритма за Ератостеново сито формулисан је следећим корацима:

1. Сваки процес креира свој део листе за означавање. Листа се састоји или од $\lceil n/p \rceil$ или од $\lfloor n/p \rfloor$ елемената.
2. Сви процеси извршавају $k = 2$, јер сваки процес мора да зна колико је k . Ово је пример понављања посла, али је операција тривијална.
3. Понављај
 - (a) Сваки процес означава умношке k у свом блоку (између k^2 и n). Морамо да одредимо први такав умножак у блоку, а после тога само означавамо сваки k -ти.
 - (б) Процес 0 налази најмањи неозначен број већи од k .
 - (в) Процес 0 емитује нову вредност k свим осталим процесима.
- све док није $k^2 > n$.
4. Неозначени бројеви су прости.
5. Редукција да одредимо колико простих бројева има.

`MPI_Bcast` је функција која омогућава процесу да емитује једну или више вредности истог типа свим осталим процесима у датом комуникатору:

```
int MPI_Bcast (void *buffer, int count,
                MPI_Datatype datatype, int root,
                MPI_Comm comm)
```

- `count` – колико елемената се емитује
- `buffer` – адреса првог податка за емитовање; сви подаци који се емитују треба да буду у непрекидном блоку меморије

- `datatype` – MPI тип података који се емитују
- `root` – ранг процеса који врши емисију
- `comm` – комуникатор који учествује у овој колективној комуникацији

У нашем случају, процес 0 емитује један цели број k свим осталим процесима:

```
MPI_Bcast (&k, 1, MPI_INT, 0, MPI_COMM_WORLD)
```

У случају да се колективне комуникације емисије и редукције желе представити на графу задатак/канал типа, канали служе за емисију у једном смеру и за редукцију у другом смеру.

5.2.4 Анализа перформанси

Нека χ представља време потребно да се означи елемент низа, укључујући додељивање јединице елементу, инкрементирање петље и проверу да ли је петља завршена. Време потребно за извршавање секвенцијалног алгоритма је онда $\chi n \ln \ln n$. Пошто се само један податак емитује у свакој итерацији, свака емисија траје $\lambda \lceil \log_2 p \rceil$. Број простих бројева између 2 и n је око $\frac{n}{\ln n}$. Онда је број итерација приближно $\frac{\sqrt{n}}{\ln \sqrt{n}}$. Дакле, очекивано време извршавања паралелног алгоритма је:

$$T_p = \chi(n \ln \ln n)/p + (\sqrt{n}/\ln \sqrt{n}) \lambda \lceil \log_2 p \rceil.$$

На Листингу 5.1 дата је имплементација алгоритма у програмском језику C.

Листинг 5.1: Имплементација паралелног алгоритма Ератостеновог сита

```

1 /* 
2  *   Eratostenovo sito
3 */
4
5 #include "mpi.h"
6 #include <math.h>
7 #include <stdio.h>
8 #define MIN(a,b) ((a)<(b) ? (a) : (b))
9
10 int main (int argc, char *argv[])
11 {
12     int      count;          /* Brojac za lokalne proste brojeve */
13     double  elapsed_time;   /* Vreme paralelnog izvršenja */
14     int      first;         /* Indeks prvog umnoska */
15     int      global_count;  /* Globalni brojac za proste */
16     int      high_value;    /* Najvecu vrednost u ovom procesu */
17     int      i;
18     int      id;            /* Rang procesa */

```

```

19     int      index;          /* Indeks tekuceg prostog broja */
20     int      low_value;      /* Najniza vrednost u ovom procesu */
21     char    *marked;         /* Porcija 2,...,'n' */
22     int      n;              /* Prosejavanje se vrsti za 2, ..., 'n' */
23     int      p;              /* Broj procesa */
24     int      proc0_size;     /* Duzina niza procesa sa rangom 0 */
25     int      prime;          /* Tekuci prost koji se koristi za
26                                prosejavanje */
27     int      size;           /* Koliko elemenata sadrzi 'marked' */
28
29     MPI_Init (&argc, &argv);
30
31     /* Pokreni tajmer */
32
33     MPI_Comm_rank (MPI_COMM_WORLD, &id);
34     MPI_Comm_size (MPI_COMM_WORLD, &p);
35     MPI_Barrier(MPI_COMM_WORLD);
36     elapsed_time = -MPI_Wtime();
37
38     if (argc != 2) {
39         if (!id) printf ("Komandna linija: %s <m>\n", argv[0]);
40         MPI_Finalize();
41         exit (1);
42     }
43
44     n = atoi(argv[1]);
45
46     /* Izracunaj velicinu niza za koji je proces zaduzen
47      kao i najnizu i najvisu vrednost indeksa
48      za proces. */
49
50     low_value = 2 + id*(n-1)/p;
51     high_value = 1 + (id+1)*(n-1)/p;
52     size = high_value - low_value + 1;
53
54     /* Proveri da li su svi prosti brojevi koji se koriste
55      za prosejavanje u okviru procesa 0 */
56
57     proc0_size = (n-1)/p;
58
59     if ((2 + proc0_size) < (int) sqrt((double) n)) {
60         if (!id) printf ("Previše procesa!\n");
61         MPI_Finalize();
62         exit (1);
63     }
64
65     /* Alociraj niz za ovaj proces */
66
67     marked = (char *) malloc (size);
68     if (marked == NULL) {

```

```

69     printf ("Problem u alokaciji memorije!\n");
70     MPI_Finalize();
71     exit (1);
72 }
73
74 for (i = 0; i < size; i++) marked[i] = 0;
75 if (!id) index = 0;
76 prime = 2;
77 do {
78     if (prime * prime > low_value)
79         first = prime * prime - low_value;
80     else {
81         if (!(low_value % prime)) first = 0;
82         else first = prime - (low_value % prime);
83     }
84     for (i = first; i < size; i += prime) marked[i] = 1;
85     if (!id) {
86         while (marked[++index]);
87         prime = index + 2;
88     }
89     if (p > 1) MPI_Bcast (&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
90 } while (prime * prime <= n);
91 count = 0;
92 for (i = 0; i < size; i++)
93     if (!marked[i]) count++;
94 if (p > 1) MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM
95 ,
96 0, MPI_COMM_WORLD);
97 /* Zaustavi tajmer */
98 elapsed_time += MPI_Wtime();
99
100
101 /* Stampaj rezultate */
102
103 if (!id) {
104     printf ("Ukupno je %d prostih brojeva manjih ili jednakih %d\n"
105             ", "
106             global_count, n);
107     printf ("SITO (%d) %10.6f\n", p, elapsed_time);
108 }
109 MPI_Finalize ();
110 return 0;
111 }
```

5.3 Унапређења

Перформансе приказана имплементације се могу значајно унапредити, и то помоћу три приступа: брисањем парних бројева, елиминисањем емитовања и реорганизацијом петљи у циљу боље употребе процесорског кеша.

5.3.1 Брисање парних бројева

Пошто је 2 једини парни прост број, нису ни потребне boolean вредности у низу за означавање за парне бројеве. У низу преостају само непарни бројеви. Ово полови количину потребне меморије и дуплира брзину означавања умножака одређеног простог броја. Са овом променом, време извршавања секвенцијалног алгоритма постаје

$$\frac{\chi(n \ln \ln n)}{2},$$

а време извршавања паралелног алгоритма:

$$T_p = \chi(n \ln \ln n)/(2p) + (\sqrt{n} / \ln \sqrt{n}) \lambda \log_2 p.$$

5.3.2 Елиминисање емисије

У свакој итерацији, један процес налази нову вредност k и еmitује је осталим процесима. Тај корак се понавља $\sqrt{n} / \ln \sqrt{n}$ пута. Ова комуникација може да се елиминише ако сваки задатак сам за себе налази нову вредност k . Задатак 0 контролише вредности од 2 до \sqrt{n} . Да би и остали задаци могли да налазе k , морају и они да имају копије ових вредности са почетка низа.

Дакле, сваки задатак ће, поред својих n/p бројева, имати одвојени низ који садржи бројеве $3, 5, 7, \dots, \lfloor \sqrt{n} \rfloor$. Пре налажења простих бројева од 3 до n , сваки задатак ће секвенцијалним алгоритмом да нађе просте бројеве од 3 до $\lfloor \sqrt{n} \rfloor$. Сви ће имати све што им треба, па више нема потребе за емисијом. Међутим, треба нагласити да елиминација емисије убрзава алгоритам само ако важи да је:

$$\begin{aligned} \frac{\sqrt{n}}{\ln \sqrt{n}} \lambda \lceil \log_2 p \rceil &> \chi \sqrt{n} \ln \ln \sqrt{n} \\ \frac{\lambda \lceil \log_2 p \rceil}{\ln \sqrt{n}} &> \chi \ln \ln \sqrt{n} \\ \lambda &> \frac{\chi \ln \ln \sqrt{n} \cdot \ln \sqrt{n}}{\lceil \log_2 p \rceil}. \end{aligned}$$

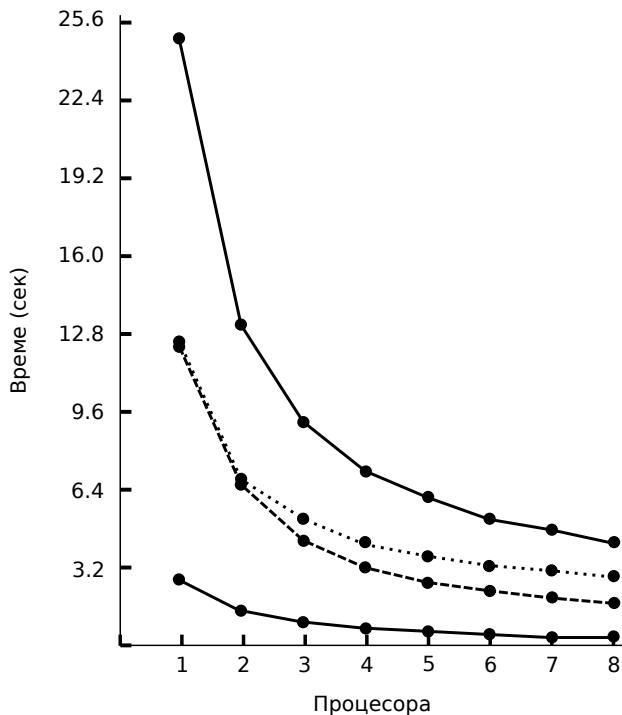
Очекивано време извршавања паралелног алгоритма је сада:

$$T_p = \chi \left(\frac{n \ln \ln n}{2p} + \sqrt{n} \ln \ln \sqrt{n} \right) + \lambda \log_2 p.$$

5.3.3 Реорганизовање петљи

Сваки процес маркира широко расуте елементе веома великог низа, што доводи до великог броја промашаја кеша. Срце алгоритма чине две петље. Спољашња петља иде од 3 до $\lfloor \sqrt{n} \rfloor$, а унутрашња кроз део бројева између 3 и n који је додељен датом процесу. Маркирамо све умношке једног простог броја, па све умношке следећег итд. Док ми маркирамо све умношке једног, пролазећи кроз цео један велики низ, и док се вратимо на почетак низа, почетне вредности низа се више не налазе у кешу.

Међутим, ако бисмо заменили спољашњу и унутрашњу петљу, можемо да повећамо број погодака кеша. Дакле, идеја је да попунимо кеш секцијом низа, да у тој секцији нађемо све умношке простих бројева мањих од $\lfloor \sqrt{n} \rfloor$, пре него што пређемо на следећу секцију.



Слика 5.2: Време извршења различитих верзија паралелног алгоритма

Дијаграм на Слици 5.2 демонстрира колико које унапређење утиче на перформансе паралелног програма. Очигледно је да је последња уведена оптимизација, реорганизација петљи, довела до највећег скока у перформансама програма. Конкретно, програм са укљученом овом оптимизацијом на једном процесору ради брже него неоптимизовани програм на 8 процесорима. Овај илустративан пример

указује да паралелизација није и не може бити једини алат за побољшање перформанси. Пре паралелизације увек треба размотрити све остале методе оптимизације.

Глава 6

Флојдов алгоритам

6.1 Секвенцијални алгоритам и извори паралелелизма

Флојдов алгоритам за дати тежински усмерени граф налази дужине најкраћих путева између сваког пара чвррова. Дужина пута је одређена тежинама грана, а не бројем чвррова кроз које се пролази. Тежински усмерени граф на рачунару представљамо матрицом суседства, јер она омогућава константно време приступа свакој ивици. За n чвррова имамо матрицу димензија $n \times n$. Елемент матрице (i, j) представља тежину ивице од чвора i до чвора j . Тежина гране графа за непостојеће ивице обележена је знаком ∞ . Решење ће такође бити матрица димензија $n \times n$, која садржи најкраће путеве између сваког пара чвррова.

Комплексност Флојдовог алгоритма: $\theta(n^3)$

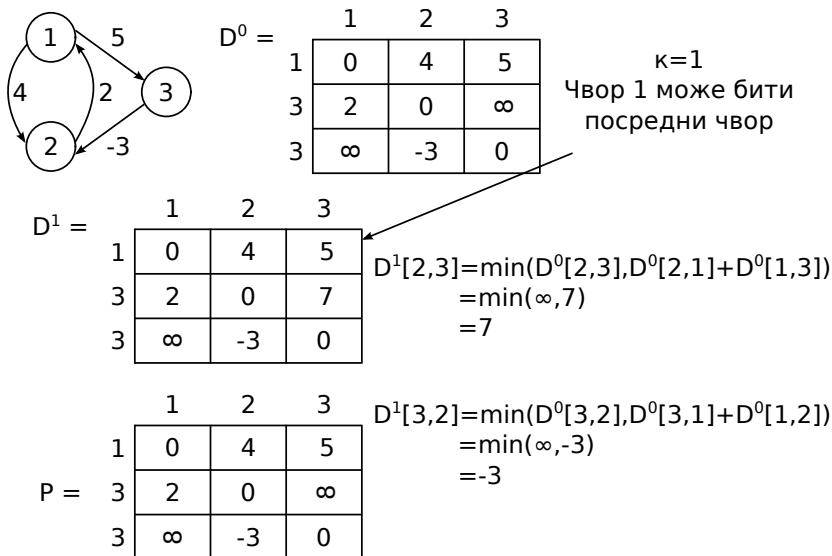
Улаз: n – број чвррова, $a[0 \dots n - 1, 0 \dots n - 1]$ – матрица суседства

Излаз: Трансформисано a које садржи дужине најкраћих путева

```
for k=0 to n-1
    for i=0 to n-1
        for j=0 to n-1
            a[i,j] = min(a[i,j], a[i,k]+a[k,j])
        end for
    end for
end for
```

На почетку сваке итерације имамо најкраће путеве од i до j , од i до k и од k до j кроз чврлове $0, 1, \dots, k - 1$. Онда проверавамо да ли нам је краће да од i до j идемо преко k . Принцип је објашњен на Слици 6.1.

Сада ћемо се позабавити имплементацијом у програмском језику С. Динамичко алоцирање једнодимензионог низа у програмском језику С врши се на следећи начин:



Слика 6.1: Флојдов алгоритам

```
int *A;
A=(int*) malloc(n*sizeof(int));
```

Програмски језик С посматра дводимензиони низ као низ низова, тј. континуалност у меморији није загарантована. Ако резервацију меморије урадимо овако:

```
int **B;
B=(int**) malloc(m*sizeof(int*));
for (i=0; i<m; i++)
    B[i]=(int*) malloc(n*sizeof(int));
```

постоји реална могућност да врсте матрице буду расштркане у меморији. Ово нам не одговара, јер хоћемо да елементи матрице заузимају непрекидни блок у меморији, да бисмо могли да пошаљемо неколико врста матрице у једној MPI поруци. Прави начин за осигурување непрекидности блока је следећи:

```
int **B, *Bstorage, i;
...
Bstorage = (int*) malloc(m*n*sizeof(int));
B = (int**) malloc(m*sizeof(int*));
for (i=0; i<m; i++)
    B[i] = &Bstorage[i*n];
```

6.2 Кораци Фостерове методологије

Пратећи Фостерову методологију, поставља се питање да ли за корак партиципонисања одабрати доменску или функционалну декомпозицију. У алгоритму се исти израз извршава n^3 пута. Дакле, пошто је у питању један једини израз, нема функционалног паралелизма. Зато приступамо доменској декомпозицији – делимо матрицу на њених n^2 елемената и повежемо по примитивни задатак са сваким елементом.

6.2.1 Комуникација

За ажурирање елемента $a[i, j]$, потребни су елементи $a[i, k]$ и $a[k, j]$. На пример, за елемент $a[3, 4]$, када је $k = 1$, потребне су вредности $a[3, 1]$ и $a[1, 4]$. За одређену вредност k , елемент $a[k, m]$ је потребан свим задацима у колони m . Током k -те итерације сваки задатак у реду k емитује свој елемент задацима у истој колони. За одређену вредност k , елемент $a[m, k]$ је потребан свим задацима у реду m . Током k -те итерације, сваки елемент у колони k се емитује свим задацима у истом реду.

Поставља се питање да ли сви елементи могу симултанско да се ажурирају? За ажурирање елемента $a[i, j]$ потребни су елементи $a[i, k]$ и $a[k, j]$. Ипак, не морамо прво њих да израчунамо јер се у k -тој итерацији $a[i, k]$ и $a[k, j]$ не мењају због:

$$\begin{aligned} a[i, k] &= \min(a[i, k], a[i, k] + a[k, k]) \\ a[k, j] &= \min(a[k, j], a[k, k] + a[k, j]) \end{aligned}$$

Дакле, вредности не могу да се смање, па остају исте, што доводи до повољне околности да нема зависности. У свакој итерацији извршимо емитовање, па онда симултанско ажуририрамо све елементе.

6.2.2 Агломерација и мапирање

Код доношења одлуке о начину агломерације и мапирања користимо стабло одлучивања. Број задатака је статичан, комуникација је структурирана, време израчунавања по задатку је константно, па је стратегија за агломерацију задатака да минимизујемо комуникацију и да креирамо један агломеризовани задатак по MPI процесу. Агломерација може да споји све задатке у истом реду, или све задатке у истој колони.

Ако користимо **агломерацију по врстама** (енг. *rowwise*), елиминишишемо емисију између задатака који припадају истој врсти. Остаје да, током сваке итерације, један задатак емитује n елемената свим осталим задацима. Свака емисија захтева $\lceil \log_2 p \rceil (\lambda + \frac{n}{\beta})$ времена.

Ако се одлучимо за **агломерацију по колонама** (енг. *columnwise*), елиминишишемо емисију између задатака у истој колони. Остаје да, током сваке итерације,

један задатак еmitује n елемената свим осталим задацима. Свака емисија захтева $\lceil \log_2 p \rceil (\lambda + \frac{n}{\beta})$ времена.

Као што можемо видети, и један и други приступ захтевају исто време за извршење. Међутим, у програмском језику С се матрице чувају у редном поретку (први ред, па други ред итд.). Зато бирамо агломерацију по врстама, јер ћемо на тај начин једноставније учитати матрицу (ред по ред) и исписати је.

Унос и испис матрице решава се тако што један процес чита матрицу са улаза и дистрибуира је свим осталим процесима. Најбоље је да то буде процес са рангом $p - 1$. Прво, може да искористи меморију, где ће на крају сачувати и свој део матрице за баферовање редова које шаље другим процесима. Друго, процес ранга $p - 1$ **сигурно добија** $\lceil n/p \rceil$ редова, тј. ниједан процес не добија више од њега, па стога има довољно места за баферовање. Дакле, процес $p - 1$ прво чита редове који иду процесу 0, па шаље процесу 0, затим чита редове који иду процесу 1, шаље процесу 1 итд.

Нека процес 0 буде одговоран за испис, да би се вредности појавиле у тачном редоследу. Процес 0 испише своју подматрицу. Онда зове процес 1, процес 1 шаље процесу 0 своју подматрицу, процес 0 штампа подматрицу 1, процес 0 зове процес 2, процес 2 шаље своју подматрицу итд. Иако процес 0 може да одреди редослед примања без обзира на редослед слања, не допуштамо осталим процесима да пошаљу своје подматрице било кад, како не би загушили процес са рангом 0.

6.3 Блокирајуће MPI_Send и MPI_Recv, застој

Комуникација од тачке до тачке (енг. *point-to-point*) укључује пар процеса, за разлику од колективних комуникација које укључују све процесе у датом комуникатору. Када процес i пошаље поруку, може одмах да настави са радом. Међутим, када процес j треба да прими поруку, блокира се и чека да порука стварно буде примљена:

```
if (id==i)
    пошаљи поруку процесу j
else if (id==j)
    прими поруку од процеса i
```

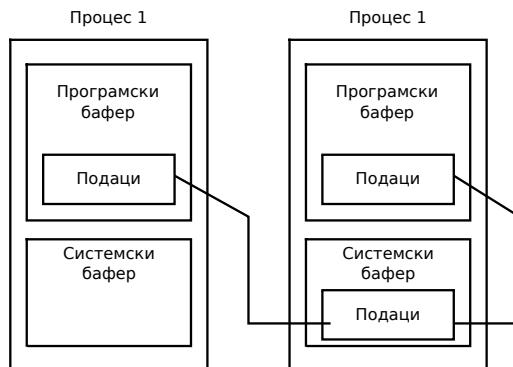
6.3.1 Функција MPI_Send

Потпис функције MPI_Send изгледа овако:

```
int MPI_Send (void *message, int count,
              MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
```

- message – почетна адреса података за слање
- count – број података
- datatype – тип података
- dest – ранг процеса који прима поруку
- tag – ознака поруке
- comm – комуникатор

`MPI_Send` спада у блокирајуће позиве, јер се блокира у случају да је бафер за поруке пун. Блокада траје док бафер поново не постане доступан. Бафер за поруке ослобађа се ако се порука пошаље у системски бафер, или ако се порука пошаље директно. Уобичајен сценарио је да *run-time* систем копира поруку у системски бафер, али не мора да буде тако, може и да је копира директно у меморију примајућег процеса. MPI стандард не дефинише начин на који се ова процедура имплементира. То је остављено творцима MPI имплементација. Начин функционисања блокирајућих позива за слање и пријем приказан је на Слици 6.2.



Слика 6.2: Начин функционисања блокирајућих MPI позива

6.3.2 Функција MPI_Recv

Потпис функције `MPI_Send` изгледа овако:

```
int MPI_Recv (void *message, int count,
              MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status status)
```

- `message` – почетна адреса где ће подаци да се усклашише
- `count` – максимални број података које је примајући процес волјан да прими
- `datatype` – тип података
- `source` – ранг процеса који шаље поруку
- `tag` – жељена ознака за поруку
- `comm` – комуникатор
- `status` – пре позива `MPI_Recv`, мора да се алоцира запис типа `MPI_Status`, `status` је показивач на тај запис.

`MPI_Recv` се блокира док се не прими порука. Када се прими, може да се приступи статусу који се састоји од следећих поља:

`status->MPI_source` – ранг процеса који шаље,
`status->MPI_tag` – ознака поруке,
`status->MPI_ERROR` – код грешке.

Source може да буде `MPI_ANY_SOURCE`, а таг може да буде `MPI_ANY_TAG`. Застој (енг. *deadlock*) настаје када је процес блокиран чекајући услов који никада неће бити испуњен. Лако је написати *send/receive* код који упада у застој. Рецимо, оба процеса примају пре него што пошаљу:

```
if (id==0) {
    MPI_Recv (& b...
    MPI_Send (& a...
} else if (id==1) {
    MPI_Recv (& a...
    MPI_Send (& b...
}
```

Процес 0 се блокира у `MPI_Recv` пре него што пошаље *a*. Процес 1 се блокира пре него што пошаље *b*. Оба су блокирана, догодио се застој. Застој такође може настати и из следећих разлога:

- Ознака у `send-y` се не поклапа са ознаком у `receive-y`.
- Процес шаље на погрешну дестинацију или прима са погрешног извора.

6.4 Комплексност и време извршења

На Листингу 6.1 дат је изворни код паралелне верзије Флојдовог алгоритма. Дискусија о комплексности и предвуђању времена извршења базира се на овој имплементацији.

Листинг 6.1: Паралелна имплементација Флојдовог алгоритма

```

1  /*
2   *      Flojdov algoritam
3   *
4   *      Data je matrica dimenzija NxN sa udaljenostima izmedju
5   *      temena. Ovaj MPI program racuna najblize rastojanje izmedju
6   *      temena.
7   *
8   *      Program demonstrira:
9   *          kako dinamicki alocirati visedimenzioni niz
10  *         kako jedan proces preuzima U/I za ostale
11  *         emisija vektora elemenata
12  *         poruke sa razmicitim oznakama (tagovima)
13  *
14  *      Michael J. Quinn
15  *
16  *      Last modification: 4 September 2002
17  */
18
19 #include <stdio.h>
20 #include <mpi.h>
21 #include "../MyMPI.h"
22
23 typedef int dtype;
24 #define MPI_TYPE MPI_INT
25
26 int main (int argc, char *argv[]) {
27     dtype** a;           /* Niz sa dva indeksa */
28     dtype* storage;     /* Lokalna porcija elemenata niza */
29     int i, j, k;
30     int id;             /* Rang procesa */
31     int m;              /* Ukupno vrsta */
32     int n;              /* Ukupno kolona */
33     int p;              /* Ukupno procesa */
34     double time, max_time;
35
36     void compute_shortest_paths (int, int, int**, int);
37
38     MPI_Init (&argc, &argv);
39     MPI_Comm_rank (MPI_COMM_WORLD, &id);
40     MPI_Comm_size (MPI_COMM_WORLD, &p);
41
42     read_row_striped_matrix (argv[1], (void *) &a,
43                             (void *) &storage, MPI_TYPE, &m, &n, MPI_COMM_WORLD);

```

```

44     if (m != n) terminate (id, "Matrix must be square\n");
45
46
47 /* 
48  print_row_striped_matrix ((void **) a, MPI_TYPE, m, n,
49   MPI_COMM_WORLD);
50 */
51 MPI_Barrier (MPI_COMM_WORLD);
52 time = -MPI_Wtime();
53 compute_shortest_paths (id, p, (dtype **) a, n);
54 time += MPI_Wtime();
55 MPI_Reduce (&time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0,
56   MPI_COMM_WORLD);
57 if (!id) printf ("Floyd, matrix size %d, %d processes: %6.2f
58   seconds\n",
59   n, p, max_time);
60 /* 
61  print_row_striped_matrix ((void **) a, MPI_TYPE, m, n,
62   MPI_COMM_WORLD);
63 */
64 }
65
66 void compute_shortest_paths (int id, int p, dtype **a, int n)
67 {
68     int i, j, k;
69     int offset; /* Lokalni indeks vrste koja se emituje */
70     int root; /* Počes koji kontrolise vrstu koja se emituje */
71     int* tmp; /* Privremeno smesta vrstu za emisiju */
72
73     tmp = (dtype *) malloc (n * sizeof(dtype));
74     for (k = 0; k < n; k++) {
75         root = BLOCK_OWNER(k,p,n);
76         if (root == id) {
77             offset = k - BLOCK_LOW(id,p,n);
78             for (j = 0; j < n; j++)
79                 tmp[j] = a[offset][j];
80         }
81         MPI_Bcast (tmp, n, MPI_TYPE, root, MPI_COMM_WORLD);
82         for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
83             for (j = 0; j < n; j++)
84                 a[i][j] = MIN(a[i][j],a[i][k]+tmp[j]);
85     }
86     free (tmp);
87 }
```

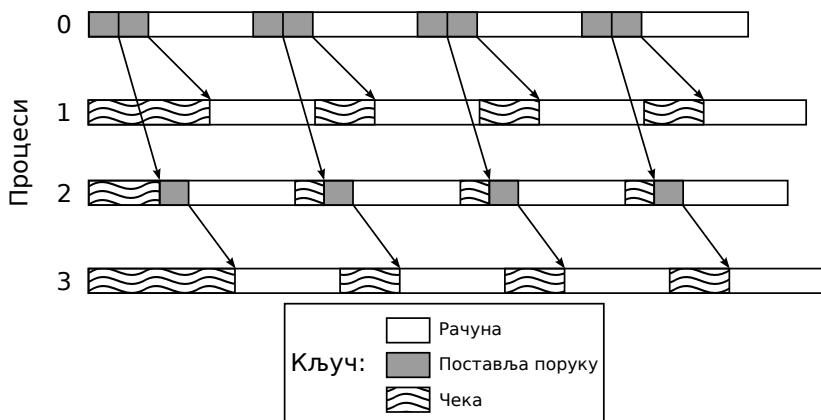
Унутрашња петља, тј. она која ажурира један ред матрице A , има комплекност $\theta(n)$, као и у секвенцијалном програму. Пошто имамо блок декомпозицију, средња петља може имати највише $\lceil n/p \rceil$ итерација. Комплексност две унутрашње петље је $\theta(n^2/p)$. Пре средње петље иде емисија. Прослеђивање поруке

дужине n од једног процесора до другог има комплексност $\theta(n)$. Пошто емисија на p процесора захтева $\lceil \log_2 p \rceil$ корака, комплексност емисије у свакој итерацији је $\theta(n \log_2 p)$. Спољашња петља се извршава n пута, па је комплексност целог алгоритма $\theta(n^3/p + n^2 \log_2 p)$.

Ако говоримо о **времену извршења** паралелног алгоритма, први елемент који треба узети у обзир је n емисија. За сваку је потребно $\lceil \log_2 p \rceil$ комуникационих корака. У сваком кораку се прослеђује порука дужине $4n$ байтова (n integer-а од по 4 байта). Очекивано време комуникације је $n \lceil \log_2 p \rceil \left(\lambda + \frac{4n}{\beta} \right)$. Ако је χ време потребно да се ажурира један елемент, време израчунавања се може проценити као $n^2 \lceil n/p \rceil \chi$. Збирно гледано, време извршења паралелног алгоритма ће бити:

$$T_p = n^2 \lceil n/p \rceil \chi + n \lceil \log_2 p \rceil \left(\lambda + \frac{4n}{\beta} \right).$$

Међутим, стварно време извршења је нешто мање, зато што се рачунања и кому-



Слика 6.3: Преклапање рачунања и комуникације код Флојдовог алгоритма

никације преклапају, као што је приказано на Слици 6.3. Када процес иницира слање поруке, може да почне да ажурира своје редове матрице док се порука још увек шаље. У првој итерацији, један процес ће морати да чека $\lceil \log_2 p \rceil \left(\lambda + \frac{4n}{\beta} \right)$ времена да прими поруку, јер је на самом почетку, па нема шта да рачуна. Постоји тога се слања порука и израчунавања у потпуности преклапају, једино што је процесу потребно $\lceil \log_2 p \rceil \lambda$ времена да иницира поруке за емисију. Да би комуникација била у потпуности преклопљена са израчунавањем (за велико n), мора да важи следеће:

$$\lceil \log_2 p \rceil \frac{4n}{\beta} < \lceil n/p \rceil n \chi.$$

Време извршавања паралелног алгоритма је, стога:

$$T_p = n^2 \lceil n/p \rceil \chi + n \lceil \log_2 p \rceil \lambda + \lceil \log_2 p \rceil \frac{4n}{\beta}.$$

Глава 7

Анализа перформанси

Већ смо делимично разматрали могућности да се предвиди време извршења паралелног програма, ако је улаз величина проблема и број процесора који учествују у извршењу. Поред тога, треба извршити и анализу перформанси већ постојећег паралелног програма и на основу ње размотрити могућности побољшања и евентуална ограничења. Ово поглавље се бави обема темама, уводећи одговарајуће квантитативне мере.

7.1 Формула за убрзање и ефикасност

Убрзање(*speedup*) је однос између секвенцијалног и паралелног времена извршавања. Формулa убрзања је:

$$\text{Убрзање} = \frac{\text{Трајање секвенцијалног извршења}}{\text{Трајање паралелног извршења}}.$$

Дакле, убрзање је неименован број који даје процену колико пута је паралелни програм бржи од секвенцијалног. Трајање операција у паралелном алгоритму се може сврстати у три категорије:

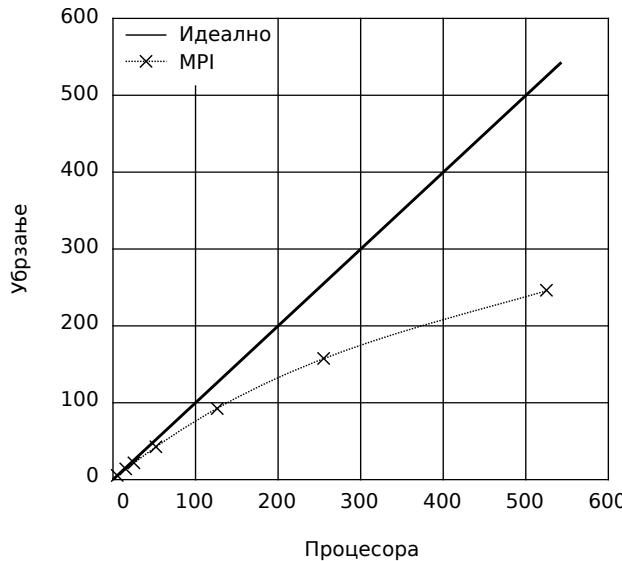
- $\sigma(n)$ – колико трају израчунавања која морају да се изврше секвенцијално
- $\varphi(n)$ – колико трају израчунавања која се потенцијално могу паралелизовати
- $\kappa(n, p)$ – колико дugo траје **паралелни додатак**. У паралелни додатак спадају редундантна израчунавања и интерпроцесна комуникација. Ови елементи не постоје у секвенцијалном програму.

Са n је обележена величина проблема који се решава (нпр. дужина низа), а са p број процесора који учествују у извршењу. Убрзање се обележава са $\psi(n, p)$.

Секвенцијални програм извршава једну операцију у тренутку и не захтева комуникацију, па се изврши за време $\sigma(n) + \varphi(n)$. Паралелни програм, у најбољем случају, паралелни део израчунавања дели савршено између p процесора, тј. $\frac{\varphi(n)}{p}$. Секвенцијални део не добија ништа паралелизацијом, тј. остаје на $\sigma(n)$. Такође, морамо додати време за међупроцесну комуникацију и редундантна израчунавања:

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p} + \kappa(n, p)}. \quad (7.1)$$

Знак мање или једнако (\leq) стоји услед претпоставке о савршеном дељењу паралелног дела међу учествујућим процесима, које није могуће постићи у свакој прилици. Додавање процесора смањује време израчунавања, али повећава комуникацију, па због тога крива изгледа као на Слици 7.1.



Слика 7.1: Убрзање паралелног програма у зависности од броја процесора

Ефикасност паралелног програма је мера искоришћења процесора. Дефинисана је као убрзање подељено бројем коришћених процесора:

$$\text{Ефикасност} = \frac{\text{Трајање секвенцијалног извршења}}{\text{Број процесора} \times \text{Трајање паралелног извршења}}$$

$$\text{Ефикасност} = \frac{\text{Убрзање}}{\text{Број процесора}}.$$

На пример, ако програм на 5 процесора ради 5 пута брже, ефикасност је максимална и износи 1. Ако на 5 процесора ради 4 пута брже, ефикасност је мања и

износи $\frac{4}{5}$. Озанака за ефикасност је $\varepsilon(n, p)$, где је $0 \leq \varepsilon(n, p) \leq 1$. Рашиљавањем формуле добија се:

$$\varepsilon(n, p) \leq \frac{\sigma(n) + \varphi(n)}{p(\sigma(n) + \frac{\varphi(n)}{p} + \kappa(n, p))} = \frac{\sigma(n) + \varphi(n)}{p\sigma(n) + \varphi(n) + p\kappa(n, p)}. \quad (7.2)$$

7.1.1 Амдалов закон

Амдалов закон је једна од најчешће коришћених мера код паралелног рачунарства. Добија се тако што се из формуле (7.1) занемари паралелни додатак $\kappa(n, p)$:

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p} + \kappa(n, p)} \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p}}. \quad (7.3)$$

Нека f представља део кода који мора да се изврши секвенцијално:

$$\begin{aligned} f &= \frac{\sigma(n)}{\sigma(n) + \varphi(n)}, \quad 0 \leq f \leq 1, \\ 1 - f &= \frac{\varphi(n)}{\sigma(n) + \varphi(n)}. \end{aligned}$$

Ако и бројилац и именилац једначине (7.3) поделимо са $(\sigma(n) + \varphi(n))$ добијамо:

$$\psi(n, p) \leq \frac{1}{\frac{\sigma(n)}{\sigma(n) + \varphi(n)} + \frac{\varphi(n)}{p(\sigma(n) + \varphi(n))}} = \frac{1}{f + \frac{1-f}{p}}. \quad (7.4)$$

Дакле, друга форма Амдаловог закона гласи:

$$\psi(n, p) \leq \frac{1}{f + \frac{1-f}{p}}.$$

Ако је секвенцијални програм представљен јединицом, у паралелном програму имамо секвенцијални део f и остатак $1 - f$ подељен на p процесора. Амдалов закон се заснива на претпоставци да покушавамо да решимо проблем фиксне величине што је могуће брже. Даје горњу границу убрзања које можемо да постигнемо упошљавањем одређеног броја процесора. Може да се искористи и да одреди асимптотско убрзање које се постиже како расте број процесора.

Пример 7.1. 10% кода је неизбежно секвенцијално. Колико је максимално убрзање на 8 процесора?

$$\psi \leq \frac{1}{0,1 + \frac{1-0,1}{8}} = \frac{1}{0,1 + \frac{0,9}{8}} = \frac{1}{0,1 + 0,11} = \frac{1}{0,21} = 4,76.$$

Пример 7.2. 25% је неизбежно секвенцијално. Колико је максимално убрзање?

$$\lim_{p \rightarrow \infty} \frac{1}{0,25 + \frac{1-0,25}{p}} = \frac{1}{0,25} = 4.$$

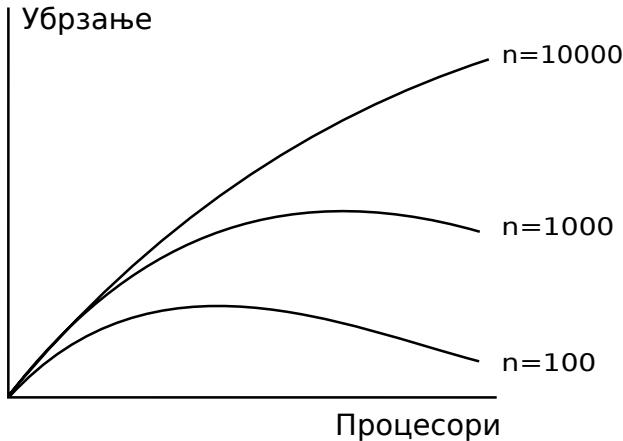
7.1.2 Ограничења Амдаловог закона

Ограничења овако формулисаног Амдаловог закона су следећа:

1. Игнорише паралелни додатак $\kappa(n, p)$ – прецењује убрзање.
2. Претпоставља да је f константно – потцењује убрзање.

7.1.3 Амдалов ефекат

Ако посматрамо бројилац разломка у једначини (7.1), намеће се чињеница да $\varphi(n)$ обично носи већу комплексност од $\sigma(n)$ и $\kappa(n, p)$. Унутар $\sigma(n)$ и $\kappa(n, p)$ улазе рутине за учитавање, испис, колективне комуникације и комуникације од тачке-до-тачке, за које је карактеристична линеарна или логаритамска комплексност. У $\varphi(n)$ је обично садржан главни део алгоритма, са вишом својственом комплексношћу. Консеквенца ове чињенице је да увећавањем проблема (када n расте) време израчунавања расте брже него време паралелног додатка. Стога, за фиксни број процесора p , убрзање је обично растућа функција величине проблема. Ова појава се зове Адалов ефекат. Последица Амдаловог ефекта је да је паралелизација проблема који су обимнији има више смисла него паралелизација проблема мањег обима, што је илустровано на Слици 7.2.



Слика 7.2: Убрзање

7.2 Густавсон-Барсисов закон, скалирано убрзање

Фокус Амдаловог закона је минимизација времена извршења паралелног израчунавања. Амдалов закон третира величину проблема као константу и пока-

зује како се време извршења смањује како број процесора расте.

Међутим, често желимо да употребимо брже рачунаре да бисмо решили веће инстанце проблема за исто време. Такође, понекад и немамо секвенцијалну верзију програма да бисмо могли да проценимо вредност f у једначини (7.4). Шта ако третирамо време као константу и пустимо да величина проблема расте са бројем процесора? Повећање броја процесора нам омогућава да увећамо проблем који решавамо, смањимо неизбежно секвенцијалну фракцију израчунавања и да, следствено, повећамо убрзање.

Нека s означава фракцију времена проведеног у извршавању секвенцијалних операција у паралелном програму. Фракција времена за потенцијално паралелне операције у паралелном програму је онда $1 - s$. Вредност s је константна без обзира на вредност p , јер се са p мења и n . Дакле, имамо да је:

$$s = \frac{\sigma(n)}{\sigma(n) + \frac{\varphi(n)}{p}}, \quad 1 - s = \frac{\frac{\varphi(n)}{p}}{\sigma(n) + \frac{\varphi(n)}{p}},$$

Ако и бројилац и именилац израза (7.3) поделимо са $\sigma(n) + \frac{\varphi(n)}{p}$, добијамо:

$$\begin{aligned} \psi(n, p) &\leq \frac{\sigma(n)}{\sigma(n) + \frac{\varphi(n)}{p}} + \frac{\varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p}} \\ \psi(n, p) &\leq s + (1 - s)p = s + p - sp \\ \psi(n, p) &\leq p + (1 - p)s. \end{aligned}$$

Како је $p + (1 - p)s = p - (p - 1)s$, имамо да је убрзање на p процесора p (савршено убрзање) умањено за време за које се на $p - 1$ процесора не извршава ништа, док се на једном једином процесору извршава секвенцијални део. Дакле, Густавсон-Барсисов закон гласи:

$$\psi(n, p) \leq p + (1 - p)s. \quad (7.5)$$

Амдалов закон полази од секвенцијалног програма, док Густавсон-Барсисов закон полази од паралелног програма. Убрзање предвиђено Густавсон-Барсисовим законом се назива још и **скаларним убрзањем**, зато што допушта да величина проблема буде растућа функција броја процесора. Основни недостатак овог закона је што претпоставља да s остаје фиксно, независно од тога колико је p , па стога прецењује убрзање.

Пример 7.3. Број процесора је 64, а време извршавања паралелног програма $220s$. 5% времена се проведе у секвенцијалном коду, тј. $s = 0,05$. Скалирано убрзање је:

$$\psi = 64 + (1 - 64)0,05 = 64 - 63 \cdot 0,05 = 64 - 3,15 = 60,85.$$

Пример 7.4. Која је максимална фракција времена која током извршавања паралелног програма може бити проведена у серијском коду, ако хоћемо да достигнемо убрзање од 7 на 8 процесора?

$$7 = 8 + (1 - 8)s, \quad 7 = 8 - 7s, \quad 7s = 1, \quad s = \frac{1}{7} = 0, 14.$$

7.3 Карп-Флет метрика, експериментално одређена серијска фракција

Пошто Амдалов закон и Густавсон-Барсисов закон занемарују $\kappa(n, p)$, они прецењују како убрзање, тако и скалирано убрзање. Како би превазишли овај проблем, Карп и Флет су предложили другачију метрику названу **експериментално одређена серијска фракција**. Ако је:

$$T(n, p) = \sigma(n) + \frac{\varphi(n)}{p} + \kappa(n, p)$$

време извршавања паралелног програма на p процесора, а:

$$T(n, 1) = \sigma(n) + \varphi(n),$$

време извршавања серијског програма, дефинишемо експериментално одређену серијски фракцију e као укупну количину паралелног додатка (време чекања и време комуникације), скалираног бројем процесора (минус 1) и секвенцијалним временом извршавања:

- време чекања (*idle*) – док један процесор изводи серијски део програма, $p - 1$ процесора не ради ништа, ту се изгуби време од $(p - 1)\sigma(n)$.
- време комуникације (*overhead*) – p процесора врши комуникацију, губи се време од $p\kappa(n, p)$.

Скалирање се врши да би се од апсолутне мере добио неименован број у интервалу између нуле и јединице:

$$e = \frac{(p - 1)\sigma(n) + p\kappa(n, p)}{(p - 1)T(n, 1)} \tag{7.6}$$

Израз (7.6) се може написати и на следећи начин:

$$\begin{aligned} e &= \frac{p\sigma(n) - \sigma(n) + p\frac{\varphi(n)}{p} - \varphi(n) + p\kappa(n, p)}{(p - 1)T(n, 1)} \\ e &= \frac{p\left(\sigma(n) + \frac{\varphi(n)}{p} + \kappa(n, p)\right) - (\sigma(n) + \varphi(n))}{(p - 1)T(n, 1)} \\ e &= \frac{pT(n, p) - T(n, 1)}{(p - 1)T(n, 1)}. \end{aligned}$$

Ако сада и бројилац и именилац претходног разломка поделимо са $T(n, p)$, узевши у обзир да је $\psi = \frac{T(n, 1)}{T(n, p)}$, добићемо:

$$e = \frac{p - \psi}{(p - 1)\psi} \quad (7.7)$$

Ако сада поделимо и бројилац и именилац са ψp , добијамо коначну формулу за рачунање експериментално одређене серијске фракције:

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}. \quad (7.8)$$

Експериментално одређена серијска фракција је корисна из два разлога:

1. Узима у обзир паралелни додатак,
2. Детектује изворе успорења и неефикасности које модел убрзања игнорише (покретање и синхронизација процеса, небалансираност,...)

За проблем фиксне величине, ефикасност паралелног израчунавања обично пада како број процесора расте. Користећи e , можемо да одредимо да ли ефикасност пада због: (1) ограничених могућности за паралелизам, или (2) пораста у алгоритамском додатку и додатку који је везан за архитектуру. Ово рашичлање је очигледно ако се има у виду структура бројиоца разломка у једначини 7.6. Тако, на основу понашања e , могу се донети одређени закључци који могу послужити за откривање узрока неефикасности у паралелном програму:

1. Ако e остаје константно док број процесора расте, узрок малог убрзања су ограничene могућности за паралелизам тј. велика фракција израчунавања које је неизбежно секвенцијално ($\sigma(n)$ у једначини (7.6)).
2. Ако e полако расте како број процесора расте, главни узрок малог убрзања је паралелни додатак, тј. члан $\kappa(n, p)$ у бројиоцу једначине (7.6).

Пример 7.5. У следећој табели су дата убрзања која постиже паралелни програм. Одредити где се може тражити узрок релативно лошег убрзања.

p	2	3	4	5	6	7	8
ψ	1,82	2,5	3,08	3,57	4,0	4,38	4,71

Решење: На основу вредности ψ може се одредити вредност e за свако p . Добијамо:

p	2	3	4	5	6	7	8
ψ	1,82	2,5	3,08	3,57	4,0	4,38	4,71
e	0,1	0,1	0,1	0,1	0,1	0,1	0,1

Дакле, вредност експериментално одређене серијске фракције је константна како број процесора расте. Закључујемо да је узрок лоших перформанси инхерентно секвенцијални део кода.

Пример 7.6. У наредној табели су дата убрзања која постиже паралелни програм. Одредити где се може тражити узрок релативно ниског убрзања паралелизацијом.

p	2	3	4	5	6	7	8
ψ	1,87	2,61	3,23	3,73	4,14	4,46	4,71

Решење: На основу вредности ψ може се одредити вредност e за свако p . Добијамо:

p	2	3	4	5	6	7	8
ψ	1,87	2,61	3,23	3,73	4,14	4,46	4,71
e	0,07	0,075	0,080	0,085	0,09	0,095	0,1

Дакле, вредност експериментално одређене серијске фракције полако расте како број процесора расте. Закључујемо да је узрок лоших перформанси паралелни додатак.

7.4 Метрика изоefикасности

Паралелни систем се дефинише као паралелни програм који се извршава на паралелном рачунару.

Скалабилност паралелног система је мера његове могућности да одржи перформансе како број процесора расте. По Амдаловом ефекту, брзина (а самим тим и ефикасност), је растућа функција величине проблема. Да бисмо одржали исти ниво ефикасности кад се додају нови процесори, морамо да повећавамо величину проблема. Наведене идеје су формализоване у релацији изоefикасности. Изоefикасност је начин да се измери скалабилност.

Почнимо са формулом за убрзање и израчунајмо укупну количину додатка:

$$\begin{aligned}\psi(n, p) &\leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p} + \kappa(n, p)} \\ &= \frac{p(\sigma(n) + \varphi(n))}{p\sigma(n) + \varphi(n) + p\kappa(n, p)} \\ &= \frac{p(\sigma(n) + \varphi(n))}{\sigma(n) + \varphi(n) + (p - 1)\sigma(n) + p\kappa(n, p)}.\end{aligned}$$

Дефинишимо $T_0(n, p)$ као укупно време које потроше сви процеси радећи посао који не постоји у секвенцијалној имплементацији проблема. Једна компонента је време које $p - 1$ процеса проводе не радећи ништа, док један процес извршава неизбежно секвенцијални код. Друга компонента је време током којег свих p процеса извршавају међупроцесне комуникације и редудантна израчунавања. Дакле,

$$T_0(n, p) = (p - 1)\sigma(n) + p\kappa(n, p).$$

Заменимо сада овај укупни додатак у једначину убрзања:

$$\psi(n, p) \leq \frac{p(\sigma(n) + \varphi(n))}{\sigma(n) + \varphi(n) + T_0(n, p)}.$$

Ефикасност се рачуна као убрзање подељено бројем процеса:

$$\varepsilon(n, p) \leq \frac{p(\sigma(n) + \varphi(n))}{\sigma(n) + \varphi(n) + T_0(n, p)} = \frac{1}{\frac{\sigma(n) + \varphi(n) + T_0(n, p)}{\sigma(n) + \varphi(n)}} = \frac{1}{1 + \frac{T_0(n, p)}{\sigma(n) + \varphi(n)}}.$$

Како је $T(n, 1) = \sigma(n) + \varphi(n)$, онда имамо да је

$$\varepsilon(n, p) \leq \frac{1}{1 + \frac{T_0(n, p)}{T(n, 1)}}.$$

Одатле лако може да се добије да је:

$$T(n, 1) \geq \frac{\varepsilon(n, p)}{1 - \varepsilon(n, p)} \cdot T_0(n, p).$$

Ако желимо константан ниво ефикасности, онда је $\frac{\varepsilon(n, p)}{1 - \varepsilon(n, p)}$ константа, па је:

$$T(n, 1) \geq C \cdot T_0(n, p). \quad (7.9)$$

Једначина (7.9) назива се **релацијом изоефикасности**. Дакле, да би се одржао исти ниво ефикасности како се број процесора повећава, n мора да се повећава тако да је задовољена неједначина (7.9). Можемо да искористимо релацију изоефикасности да одредимо распон броја процесора за који одређени ниво

ефикасности може бити одржан. Ефикасност одржавамо повећавањем величине проблема, при чиму је главно ограничење да подаци којима манипулишемо морају да стану у главну меморију. Максимална величина проблема је ограничена доступном радном меморијом. Укупна количина доступне меморије је линеарна функција броја процесора.

Ако се релација изоefикасности напише као

$$n \geq f(p),$$

а са $M(n)$ означи меморија која је потребна за проблем величине n , онда се укупна потребна меморија за решавање неког проблема може изразити и као функција броја процесора $M(f(p))$. Количина потребне меморије по једном процесору је:

$$\frac{M(f(p))}{p}. \quad (7.10)$$

Ова функција показује како меморија заузета по процесору мора да расте са бројем процесора, како би се одржала иста ефикасност. Функција $\frac{M(f(p))}{p}$ назива се **функцијом скалабилности**.

Комплексност функције скалабилности одређује распон броја процесора за који константни ниво ефикасности може бити одржан. Ако је $\frac{M(f(p))}{p} = \theta(1)$, меморијски захтеви по процесору су константни и паралелни систем је савршено скалабилан. Ако је комплексност већа, захтеви за меморијом ће расти са бројем процесора и у једном тренутку ће достићи меморијски лимит, као што је приказано на Слици 7.3.

Пример 7.7 (Редукција). Да би се добила релација изоefикасности, прво мора да се процени комплексност секвенцијалног алгоритма и комплексност паралелног додатка:

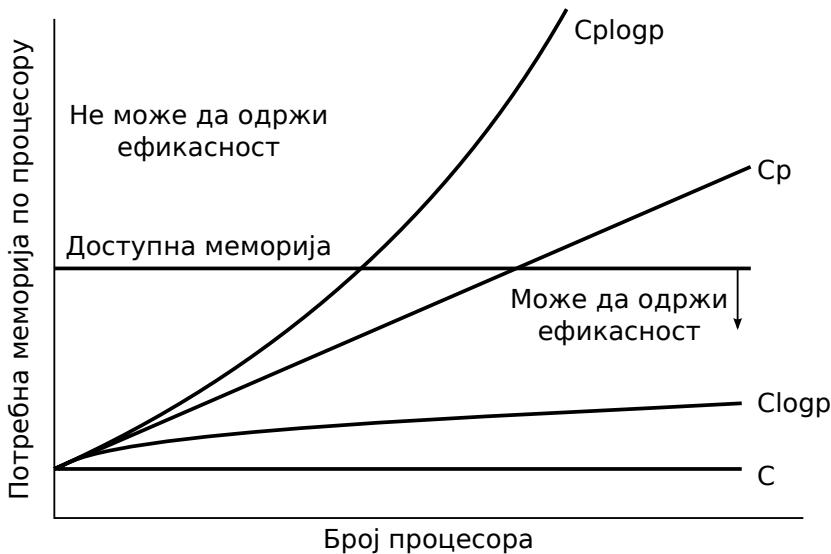
- Комплексност секвенцијалног алгоритма – $\theta(n)$
- Комплексност комуникације – $\theta(\log_2 p)$

Пошто сви процесори учествују у кораку редукције, паралелни додатак се може изразити као $T_0(n, p) = \theta(p \log_2 p)$. Сходно томе, релација изоefикасности гласи:

$$n \geq C p \log_2 p$$

Како секвенцијални алгоритам редукује n вредности, то је меморијска захтевност $M(n) = n$. Зато:

$$\frac{M(C p \log_2 p)}{p} = \frac{C p \log_2 p}{p} = C \log_2 p.$$



Слика 7.3: Функција скалабилности и меморијска ограничења

Са становишта једноставне логике такође можемо потврдити да овај резултат има смисла. Ако редукујемо n вредности на p процесора, сваки ће бити задужен за n/p вредности и учествоваће у $\lceil \log_2 p \rceil$ корака комуникације. Ако сада удвостврчимо и број вредности и број процесора, сваки процесор добиће исти број вредности као и раније, n/p . Међутим, број корака редукције ће сада порастти са $\lceil \log_2 p \rceil$ на $\lceil \log_2(2p) \rceil$. Ова чињеница имплицира да је ефикасност нешто умањена. Облик функције скалабилности то потврђује, па да би се одржала иста ефикасност, величина проблема мора да расте као $\theta(\log_2 p)$.

Пример 7.8 (Флојдов алгоритам). Комплексност секвенцијалног алгоритма и комплексност паралелног додатка износе редом:

- Комплексност секвенцијалног алгоритма – $\theta(n^3)$
- Комплексност комуникације – $\theta(n^2 \log_2 p)$

Даље:

$$\begin{aligned} T(n, 1) &= \theta(n^3) \\ T_0(n, p) &= \theta(pn^2 \log_2 p) \end{aligned}$$

Стога:

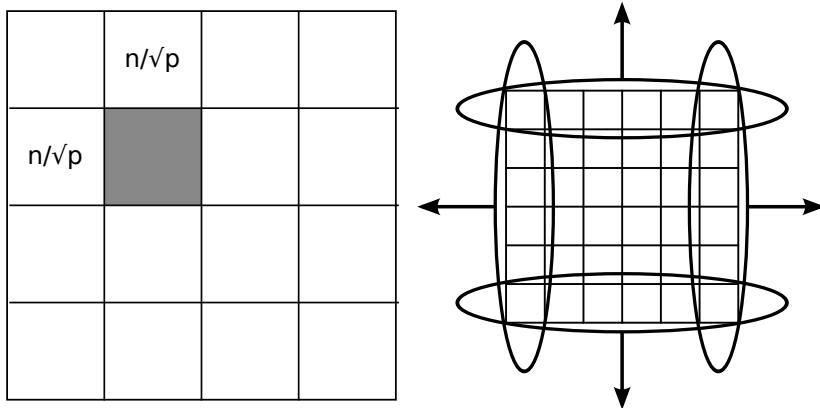
$$\begin{aligned} T(n, 1) &\geq C \cdot T_0(n, p) \\ n^3 &\geq C \cdot pn^2 \log_2 p \\ n &\geq C \cdot p \log_2 p. \end{aligned}$$

Пошто Флојдов алгоритам користи матрице, за проблем величине n потребно је n^2 меморије, $M(n) = n^2$. На основу тога, функција скалабилности износи:

$$\frac{M(f(p))}{p} = \frac{M(C \cdot p \log_2 p)}{p} = \frac{C^2 \cdot p^2 \log_2^2 p}{p} = C^2 \cdot p \log_2^2 p.$$

Овај паралелни систем има релативно лошу скалабилност.

Пример 7.9 (Коначне разлике 2Д). Метод коначних разлика за решавање парцијалних диференцијалних једначина. Проблем је представљен мрежом $n \times n$ (Слика 7.4). Сваки процесор је одговоран за подмрежу величине $(n/\sqrt{p}) \times (n/\sqrt{p})$.



Слика 7.4: Лево: Домен декомпозиција мреже за коначне разлике у 2Д; Десно: Шема комуникације за један поддомен

Током сваке итерације, сваки процесор шаље граничне вредности четворици својих суседа, па је време потребно за ове комуникације $\theta(n/\sqrt{p})$ по итерацији. Комплексност секвенцијалног алгоритма је $\theta(n^2)$ по итерацији. Дакле, $T(n, 1) = \theta(n^2)$, а $T_0(n, p) = \theta(pn/\sqrt{p})$, па је:

$$\begin{aligned} T(n, 1) &\geq C \cdot T_0(n, p) \\ n^2 &\geq C \cdot pn/\sqrt{p} \\ n &\geq C\sqrt{p}. \end{aligned}$$

Пошто је $M(n) = n^2$, функција скалабилности се рачуна као:

$$\frac{M(C\sqrt{p})}{p} = \frac{(C\sqrt{p})^2}{p} = \frac{C^2 p}{p} = C^2.$$

Комплексност је $\theta(1)$, па је овај систем савршено скалабилан.

Глава 8

Класификација документа

8.1 Опис проблема, пројектовање

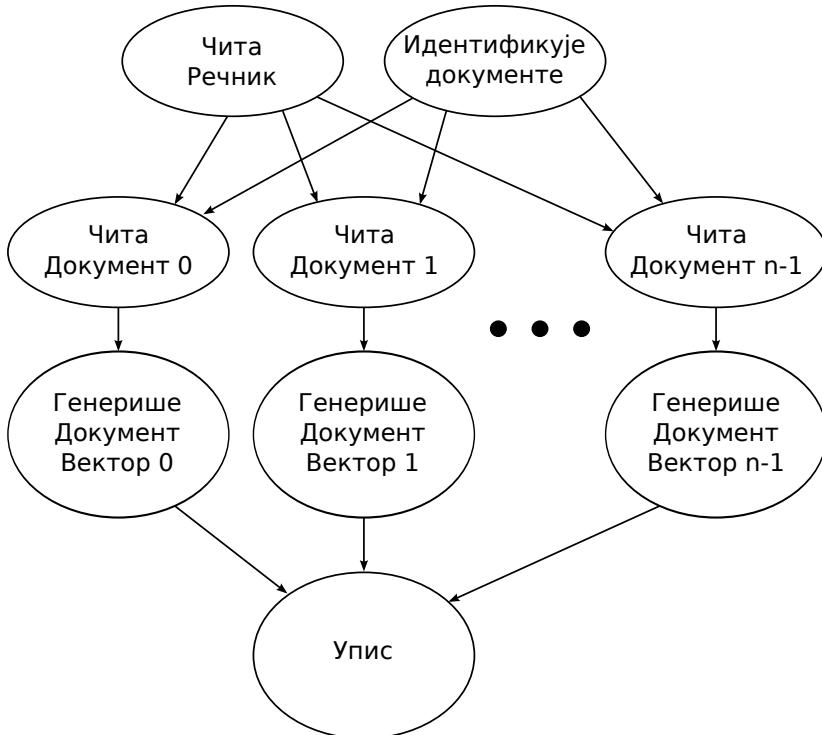
Задатак је развој паралелног програма који претражује директоријуме и поддиректоријуме са текстуалним документима (на пример .html, .txt, .tex исл.). Програм читава речник кључних речи, отвара фајлове, чита их и креира одговарајући вектор за сваки документ. Вектор има по једну димензију за сваку кључну реч и показује колико пута се у документу понавља свака кључна реч из речника. На крају се исписује матрица који садржи профилне векторе за све текстуалне фајлове које смо обрадили. Граф зависности се може илустровати шемом 8.1.

8.1.1 Партиционисање и комуникација

Читање речника и идентификација докумената не може да се обавља истовремено, док остатак алгоритма може да се паралелизује. Највећи део времена одузима читање докумената и генерирање профилних вектора. Зато правимо по два примитивна задатка за сваки документ, један за читање, а други за генерирање профилног вектора.

8.1.2 Агломерација и мапирање

Број задатака није познат за време компајлирања. Оно што је повољно за паралелизацију је чињеница да задаци не комуницирају међусобно. Међутим, време извођења сваког задатка може значајно да варира. Узрок томе је што су неки документи дужи од других, html је теже обрадити него txt итд. Дрво одлучивања дефинисано у Одељку 3.3 предлаже да мапирамо задатке на процесоре у току извршавања.



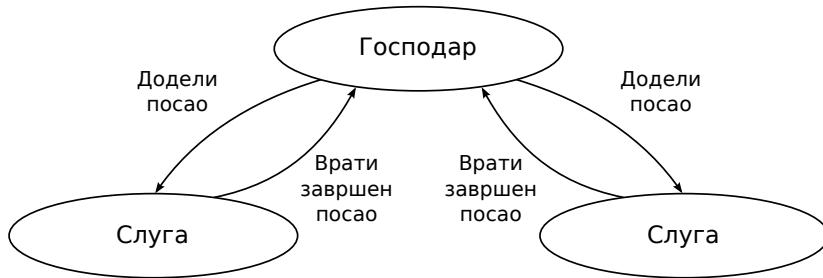
Слика 8.1: График зависности

8.1.3 Концепт господар/слуга

Алокација задатака на процесе се постиже у време извршавања алгоритмом господар/слуга. Суштина је у томе да један процес, тзв. **господар**, додељује задатке другим процесима, тзв. **слугама** и преузима резултате од њих, Слика 8.2. Овај концепт може да се посматра и као партиционисање, са алокацијом података у време извршења уместо преалокације, како је био случај у претходним задацима.

Предност оваквог приступа је што се сваком слуги у тренутку додељује само један задатак, што добро балансира оптерећење. Мана оваквог приступа је дојатна комуникација, која повећава време извршења, што имплицира и умањење убрзања.

До сада смо писали SPMD (енг. *Single Program Multiple Data*) програме, где сваки процес извршава исте функције. Код програма типа господар/слуга, господар има другачије одговорности од слуге. Обично се рано у програму догоди гранање које господара шаље на извршавање једне групе функција, а слуге на извршавање друге групе функција. Идентификовање докумената је посао за господара,



Слика 8.2: Концепт господар-слуга

пошто ће он да додељује путање до докумената слугама. Читање речника треба да обаве слуге, јер они треба да конструишу профилне векторе.

Слуга шаље поруку господару да је спреман. Господар тада додељује задатак слуги. После неког времена, слуга враћа извршен задатак господару. Сада господар може да додели том слуги други задатак. Овако господар шаље слугама задатке само ако је сигуран да су активни, јер увек прво чека поруку од њих, чак иако је у питању тренутак одмах након покретања програма. На крају, господар сакупља све векторе докумената и исписује их у фајл. Псеудокод је дат у оквиру Алгоритма 1.

Алгоритам 1 Псеудокод господара

- 1: Идентификуј документе
 - 2: Прими величину речника од слуге 0
 - 3: Алоцирај матрицу - низ вектора докумената
 - 4: **repeat**
 - 5: Прими поруку од слуге
 - 6: **if** Порука садржи вектор документа **then**
 - 7: Сачувавј вектор документа
 - 8: **end if**
 - 9: **if** Има још докумената **then**
 - 10: Пошаљи слуги име документа
 - 11: **else**
 - 12: Пошаљи слуги поруку да је завршио
 - 13: **end if**
 - 14: **until** Све слуге не заврше
 - 15: Испиши векторе у фајл
-

Псеудокод слуге дат је у оквиру Алгорима 2. Сваком слуги треба копија речника. Прва опција је да сваки од њих учита садржај фајла са речником. Друга опција је да један слуга учита речник и емитује га осталим слугама. Зависно од тога да ли је већи проток између фајл сервера и паралелног рачунара или између

чвррова паралелног рачунара, бира се једна од опција. Ми ћемо се определити за ову другу.

Алгоритам 2 Псеудокод слуге

- 1: Пошаљи први захтев за послом менаџеру
 - 2: **if** Радник 0 **then**
 - 3: Учитај речник из фајла
 - 4: Емитуј речник свим радницима (не и господару)
 - 5: **end if**
 - 6: Направи хеш табелу од речника
 - 7: **if** Радник 0 **then**
 - 8: Пошаљи величину речника господару
 - 9: **end if**
 - 10: **loop**
 - 11: Прими име фајла од господара
 - 12: **if** Примљена порука празна и означава крај **then**
 - 13: Заврши радника
 - 14: **end if**
 - 15: Прочитај документ, генериши вектор документа
 - 16: Пошаљи вектор документа господару
 - 17: **end loop**

На Слици 8.3 је приказан граф типа задатак/канал на коме се јасно види како тече комуникација између процеса који су описаны Алгоритмима 1 и 2.

8.1.4 MPI Abort

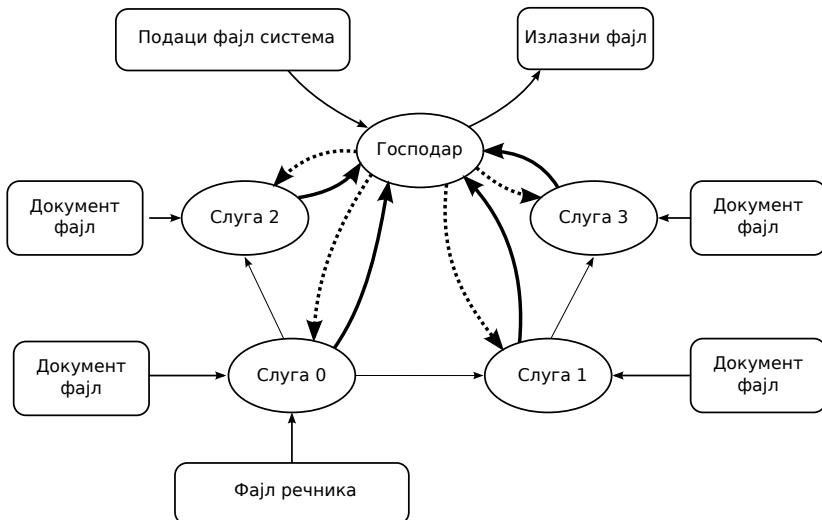
Ако гласподар не може да алокира меморију потребну да се сачувава профилни вектори, потребно је прекинути извршење целог програма. Функција MPI_Abort даје могућност једном процесу да прекине све процесе у комуникатору. Потпис ове корисне функције је:

```
int MPI_Abort (MPI_Comm comm, int error_code)
```

8.1.5 Креирање комуникатора само за раднике

Речник треба да се емитује међу слугама не укључујући господара. Емисија не укључује цео комуникатор, па нам је стога потребан нови комуникатор само за слуге. Ово се може постићи функцијом `MP_I_Comm_split`, која дели постојећи комуникатор у подгрупе:

```
int MPI_Comm_split(MPI_Comm old_comm, int partition,  
                   int new_rank, MPI_Comm *new_comm),
```



Слика 8.3: Шема интерпроцесне комуникације. Танким стрелицама је означена емисија унутар посебног комуникатора за раднике

при чemu су:

- `old_comm` - постојећи комуникатор
- `partition` - број партиције
- `new_rank` - ранг процеса у оквиру новог комуникатора
- `new_comm` - функција враћа нови комуникатор којем позивајући процес припада

У нашем случају, не желимо да господар буде члан новог комуникатора, па за њега прослеђујемо `MPI_UNDEFINED` као број партиције.

Листинг 8.1: Раздвајање комуникатора

```

1 int id;
2 MPI_Comm worker_comm;
3 if (!id) /* Gospodar */
4     MPI_Comm_split(MPI_COMM_WORLD, MPI_UNDEFINED, id, &worker_comm);
5 else /* Sluga */
6     MPI_Comm_split(MPI_COMM_WORLD, 0, id, &worker_comm);

```

8.2 Неблокирајуће MPI функције

`MPI_Send` и `MPI_Recv` су блокирајуће функције. `MPI_Send` блокира док се порука не ископира у системски бафер, или док се не пошаље директно на одредиште. Бафер одакле се шаље порука може да се препише чим се `MPI_Send` изврши. `MPI_Recv` блокира док не прими поруку у свој бафер. Поруци може да се приступи чим се `MPI_Recv` изврши.

Блокирајуће функције смањују перформансе програма. Можда не желимо да препишемо бафер са поруком за слање чим се изврши `MPI_Send`. Тада би наредне операције могле да се изврше и пре него што је слање стварно извршено, па не би имало потребе за блокирањем.

Са друге стране, иницирање примања пре него што стигне порука штеди време, јер се после порука директно копира у одредишни бафер, а не прво у системски бафер. Са `MPI_Recv` је ово тешко постићи. Ако се `recv` позове пре `send`, `recv` блокира. Ако се `send` позове пре `recv`, `send` шаље поруку у системски бафер. Ако је системски бафер пун, `send` блокира.

Неблокирајуће функције `MPI_Irecv` и `MPI_Irecv` иницирају слање и примање. Функција `MPI_Wait` експлицитно блокира процес док се комуникација не заврши. `MPI_Irecv` може да се позове чим су вредности додељене. `MPI_Irecv` може да се позове чим је бафер доступан.

Иницирање комуникације, извршавање других операција, па комплетирање комуникације штеди време на два начина. Прво, може да елиминише један корак копирања, док не користи системски бафер. Друго, дозвољава комуникационим копроцесорима (ако постоје) да извршавају комуникацију док процесори (енг. CPU) обављају задатке израчунавања.

Господар на почетку свог изршавања треба да нађе текстуалне фајлове и да прими величину речника од слуге 0. Сада, користећи неблокирајуће позиве, ове две операције могу да се преклопе.

8.2.1 Функција `MPI_Irecv`

Потпис функције `MPI_Irecv` је следећи:

```
int MPI_Irecv(void *buffer, int cnt, MPI_Datatype dtype,
              int src, int tag, MPI_Comm comm,
              MPI_Request *handle)
```

После позива ове функције не може да се приступи бафери све док се не врати позив функције `MPI_Wait`. `MPI_Irecv` враћа кроз `MPI_Request` објекат који представља иницирану комуникациону операцију. Нема `MPI_Status` објекта јер примање још није комплетирано.

8.2.2 Функција MPI_Wait

Потпис функције MPI_Wait је следећи:

```
int MPI_Wait(MPI_Request *handle, MPI_Status *status)
```

Ова функција блокира док се не заврши операција повезана са handle. Ако је та операција слање, после wait-а баферу могу да се доделе нове вредности. Ако је операција примање, после wait-а бафер може поново да се референцира и status показује на MPI_Status објекат који садржи информације о примљеној поруци.

Сада ће укратко бити описана комуникација код процеса слуга. Прво, сваки слуга мора да обавести господара да је активан. Може да иницира ово слање господару и да одмах пређе на емитовање речника и конструкцију хеш табеле. Слуга такође мора да прими путање фајлова од господара. Не зна се колико су дугачке те путање до фајлова, па би зато било добро да радник може да провери долазну поруку и одреди јој дужину пре него што је стварно прочита.

8.2.3 Функција MPI_Isend

Потпис функције MPI_Isend је следећи:

```
int MPI_Isend(void *buffer, int cnt, MPI_Datatype dtype,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *handle)
```

Бафер са поруком не може поново да се користи све док се не врати позивом функције MPI_Wait.

8.2.4 Функција MPI_Probe

Потпис функције MPI_Probe је следећи:

```
int MPI_Probe(int src, int tag, MPI_Comm comm,
              MPI_Status *status)
```

где су:

- src - ранг извора поруке;
- tag - таг долазеће поруке;
- comm - комуникатор;
- status - показивач на MPI_Status објекат.

Ова функција блокира док порука са специфицираним извором и ознаком (енг. *tag*) не постане доступна за примање. Враћа кроз *status* показивач информације о извору, ознаки и дужини поруке, али не прима стварно поруку. Питање је због чега су нам потребне информације о извору и ознаки када их ми сами задајемо. Одговор је да може да се зада MPI_ANY_SOURCE као *src* аргумент и MPI_ANY_TAG као *tag* аргумент, па онда накнадно проверити њихове вредности и на основу њих спровести одговарајуће одлуке.

8.2.5 Функција MPI_Get_count

Потпис функције MPI_Get_count је следећи:

```
int MPI_Get_count (MPI_Status *status,
                   MPI_Datatype dtype, int *cnt)
```

Прослеђујемо статус и тип елемената поруке, а кроз *cnt* добијамо број елемената у поруци. Употреба је приказана на листинзима.

Листинг 8.2: Код за господара

```
1 MPI_Irecv(&dict_size, 1, MPI_INT, MPI_ANY_SOURCE, DICT_SIZE_MSG,
            MPI_COMM_WORLD, &pending);
2 get_names(...);
3 MPI_Wait(&pending, &status);
4 ...
5 buffer = (uchar *) malloc(dict_size* sizeof(MPI_UNSIGNED_CHAR));
6 ...
7 do {
8     MPI_(buffer, dict_size. MPI_UNSIGNED_CHAR, MPI_ANY_SOURCE,
          MPI_ANY_TAG, MPI_COMM_WORLD, &status);
9     MPI_Send(file_name [assign_cnt], stelem (file_name[assign_cnt])
              +1, MPI_CHAR, src, FILE_NAME_MSG, MPI_COMM_WORLD);
10    ...
11}
```

Листинг 8.3: Код за слугу

```
1 MPI_Comm_rank(worker_comm, &worker_id);
2 MPI_Isend(NULL, 0, MPI_UNSIGNED, 0, EMPTY_MSG, MPI_COMM_WORLD, &
           pending);
3
4 if (!worker_id)
5     read_dictionary (...);
6
7 MPI_Bcast(&file_len, 1, MPI_LONG, 0, worker_comm);
8
9 if (worker_id)
10    buffer = (char *) malloc (file_len);
11
```

```

12 MPI_Bcast(buffer, file_len, MPI_CHAR, 0, worker_comm);
13 build_hash_table (...);
14
15 if (!worker_id)
16     MPI_Send (&dict_size, 1, MPI_INT, 0, DICT_SIZE_MSG,...);
17
18 for (;;) {
19     MPI_Probe(0, FILE_NAME_MSG, MPI_COMM_WORLD, &status);
20     MPI_Get_count(&status, MPI_CHAR, &name_len);
21     name = (char *) malloc (name_len);
22     MPI_Recv(name, name_len, MPI_CHAR, 0, FILE_NAME_MSG,
23               MPI_COMM_WORLD, &status);
24     make_profile (...);
25     MPI_Send(profile, dict_size, MPI_UNSIGNED_CHAR, 0, VECTOR_MSG,
26               MPI_COMM_WORLD);
27     ...

```

8.3 Додатна побољшања

Описани алгоритам може се у извесној мери побољшати. Наиме, преалокација података доводи до небалансираног оптерећења. Алоцирање података у току извршавања добро балансира оптерећење, али уводи додатне међупроцесне комуникације. У суштини, најбоље је одабрати средину, тј. определити се за додељивање по k задатака (документата) одједном.

До сада смо се бавили паралелизовањем читања документата и генерисања профилних вектора. Међутим, и њихова идентификација може одузети значајно време. У нашем решењу, ниједан документ се не обрађује док се не идентификују сви документи. Шта се догађа када поделимо задатак идентификације документата на мање јединице? Док идентификујемо документ 2, већ можемо да почнемо да читамо документ 1 итд. Док смо у једној фази обраде документа i , можемо бити у каснијим фазама обраде документата $i - 1$, $i - 2$ итд.

Овај приступ се назива цевоводом (енг. *pipeline*, Слика 1.10). Приступ цевоводу може значајно да смањи време извршења. Мана је што је сад програм у значајној мери компликованији. Сада, чим господар идентификује барем један документ и прими бар један захтев од слуге, почиње да шаље имена документата процесима. Нека је a број додељених послова, j број доступних послова и w број радника који чекају на додељивање. Алгоритам 3 објашњава како се то постиже.

8.3.1 Функција MPI_Testsome

Треба нам начин да без блокирања проверимо да ли је стигла једна или више очекиваних порука. Господар иницира неблокирајуће примање од сваког слуге.

Алгоритам 3 Решење проблема класификације користећи приступ цевовода

```

1: repeat
2:   if j>0 and w>0 then
3:     додели посао слуги
4:     j=j-1; w=w-1; a=a+1
5:   else if j>0 then
6:     обради надолазеће поруке од слуга
7:     ...
8:   else
9:     узми следећи посао
10:    j=j+1
11:  end if
12: until a=n and w=p

```

Прави низ MPI_Request објеката. Онда зове функцију MPI_Testsome, која враћа информацију колико порука је стигло.

```
int MPI_Testsome (int in_cnt, MPI_Request *handlearray,
                  int *out_cnt, int *index_array,
                  MPI_Status *status_array)
```

где су:

- in_cnt - број неблокирајућих примања које треба проверити (указни),
- handlearray - низ *handle*-ова за примања (указни),
- out_cnt - број комплетираних комуникација (изказни),
- index_array - индекси у handlearray комплетираних комуникација (изказни),
- status_array - статуси комплетираних комуникација (изказни).

Глава 9

Монте-Карло методе

Монте-Карло метод је опште прихваћен назив за широку класу компјутерских алгоритама код којих се проблем решава користећи статистичко узорковање. Примењује се често за решавање физичких и математичких проблема и најкориснији је онда када је тешко или немогуће искористити друге математичке методе. Једну варијанту Монте-Карло методе користио је француски мислилац Буфон у XVIII веку да би приближно израчунао вредност броја π . Овај експеримент је познат под називом *Буфонова иља* [11]. Тридесетих година прошлог века, Енрико Ферми, италијански физичар, опробао је Монте-Карло методу како би моделирао дифузију неутрона.

9.1 Примене Монте-Карло методе, паралелизам

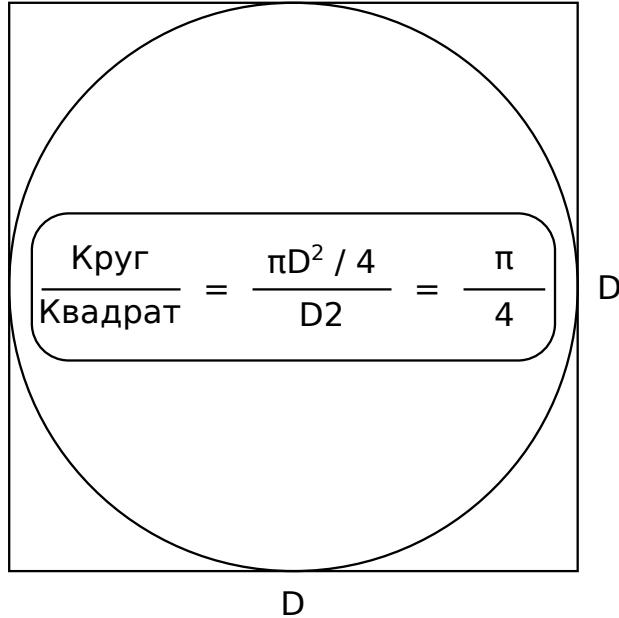
Монте Карло највећу примену налази у три одовјене класе проблема: оптимизација, нумериčка интеграција и генерисање примерака из одређене дистрибуције вероватноће. Ако говоримо о конкретним применама, на пример, ово је једини практичан начин за решавање интеграла произвольне функције у шест и више димензија. Ту је такође и предвиђање будућих вредности залиха, решавање парцијалних диференцијалних једначина, изоштравање сателитских снимака, моделовање популације ћелија, налажење апроксимације решења за НР-комплексне проблеме итд.

9.1.1 Пример рачунања броја π

Површина крга пречника D је $\frac{D^2\pi}{4}$, јер је $r = \frac{D}{2}$. Површина квадрата странице D је D^2 . Однос између ове две површине је:

$$\frac{\frac{D^2\pi}{4}}{D^2} = \frac{\pi}{4}.$$

Поставимо круг и квадрат на начин дат на Слици 9.1. Случајним избором бирамо



Слика 9.1: Рачунање броја π случајним узорковањем

тачке у оквиру квадрата. Однос између броја тачака унутар круга и броја тачака ван круга (али унутар квадрата) би требало да буде око $\frac{\pi}{4}$. Бићемо све ближи тој вредности, што је број случајних тачака већи.

Ако говоримо о имплементацији оваквог решења, комплетан круг са полу-пречником 1 има површину $r^2\pi = \pi$, па је површина четвртине круга $\frac{\pi}{4}$. Генеришими сада серију парова (x, y) где су и x и y узети из униформне случајне дистрибуције у интервалу између 0 и 1. Рачунамо фракцију f тачака које падају унутар четвртине круга, тј. тачака за које важи $x^2 + y^2 \leq 1$. Пошто је $f \approx \frac{\pi}{4}$, знамо да је $4f \approx \pi$.

Релативна грешка је, поред апсолутне грешке, начин да се квантификује квалитет процењене вредности. Што је грешка мања, процена је боља. За дату процењену вредност e и тачну вредност a , релативна грешка је $\frac{|e-a|}{a}$. Релативна грешка опада како величина узорка расте.

9.1.2 Рачунање одређеног интеграла

Дефиниција средње вредности функције $f(x)$ на интервалу $[a, b]$ гласи:

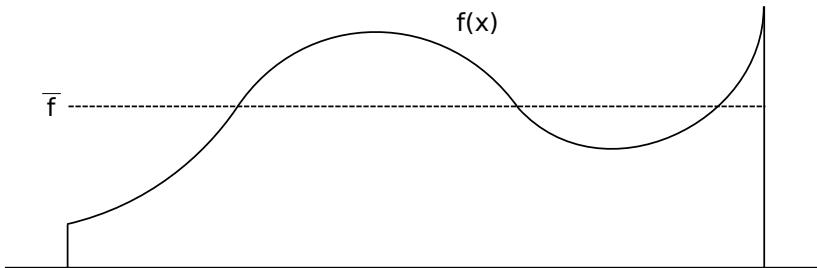
$$\int_a^b f(x)dx = (b-a)\bar{f},$$

где \bar{f} представља средњу вредност функције $f(x)$ на интервалу $[a, b]$. Монте Карло метод проценује средњу вредност функције тако што одређује вредности $f(x)$ за n тачака одабраних из униформне случајне дистрибуције над интервалом $[a, b]$ (Слика 9.2). Очекивана средња вредност је:

$$\bar{f} = \frac{1}{n} \sum_{i=0}^{n-1} f(x_i),$$

тако да горњу дефиницију можемо да трансформишимо у:

$$\int_a^b f(x)dx = (b-a)\bar{f} \approx (b-a) \frac{1}{n} \sum_{i=0}^{n-1} f(x_i).$$



Слика 9.2: Средња вредност функције на интервалу

Монте Карло је ефективан из следећа два разлога:

1. Грешка у Монте Карлу процени опада према фактору $\frac{1}{\sqrt{N}}$.
2. Стопа конвергенције је независна од димензије интегранда.

Са друге стране, код детерминистичких метода нумеричке интеграције, стопа конвергенције опада како број димензија расте. Ако имамо n одбирака по димензији, онда је укупан број одбирака $N \equiv n^d$ за d -димензиону интеграцију. Грешка се код детерминистичких метода може проценити као:

- $\propto \frac{1}{n}$ за Правило средње вредности,
- $\propto \frac{1}{n^2}$ за Трапезно правило и
- $\propto \frac{1}{n^4}$ за Симпсоново правило.

Ако знамо да се грешка Монте-Карло интеграције изражава као $\frac{1}{\sqrt{N}}$, за колико димензија Монте-Карло достиже Симпсоново правило? Процена може да се изврши на следећи начин:

$$\frac{1}{n^4} = \frac{1}{N^{\frac{4}{d}}} = \frac{1}{\sqrt{N}} \Rightarrow d \approx 8$$

За практичну примену, Монте-Карло се користи када број димензија пређе шест.

9.1.3 Паралелизам

Монте Карло методе су, по правилу, веома погодне за миграцију на паралелне рачунарске системе. Паралелни Монте Карло програми, по правилу, поседују занемарљиву количину међупроцесне комуникације. Тада p процесора може да буде употребљено на два начина:

1. да процени вредност p пута брже, или
2. да смањи грешку процене \sqrt{p} пута, јер је $\frac{1}{\sqrt{np}} = \frac{1}{\sqrt{p}} \cdot \frac{1}{\sqrt{n}}$.

Један од главних изазова у развоју паралелних Монте Карло метода је развој квалитетних паралелних генератора случајних бројева. Пре него што се позађавимо паралелним генераторима случајних бројева, потребно је да се уопште упознамо са начином на који се случајни бројеви алгоритамски генеришу.

9.2 Секвенцијални генератори случајних бројева

Генератори случајних бројева на савременим рачунарима су, у ствари, генератори псевдо-случајних бројева, зато што су њихове операције детерминистичке, и самим тим су секвенце које производе предвидиве. У најбољем случају, ове секвенце су разумна апроксимација праве случајне секвенце. Следи десет особина идеалног генератора случајних бројева:

1. Униформна дистрибуција - сваки могући број је једнако вероватан,
2. Бројеви су неповезани,
3. Нема циклуса, бројеви се никад не понављају,
4. Задовољава сваки статистички тест случајности,

5. Изводљив је,
6. Независан од машине, генератор прави исту секвенцу на било ком рачунару,
7. Промена иницијалне *seed* вредности мења секвенцу,
8. Лако се дели на много независних подсеквенција,
9. Брз је,
10. Захтева ограничenu количину меморије.

Ниједан реални генератор случајних бројева не испуњава све захтеве. Пошто се рачунари ослањају на аритметику са коначном прецизношћу, следи да оперишемо коначним бројем стања, па ће у једном тренутку бројеви неизбежно почети да се понављају, тј. доћи ће до појаве циклуса. **Период генератора** дефинише се као дужина његовог циклуса.

Пошто захтевамо да генератор буде изводљив, следи да бројеви не могу бити потпуно неповезани. Често морамо да тргујемо између брзине генератора и квалитета секвенце бројева које генерише. Пошто је време потребно за генерисање случајног броја само мали део укупног времена израчунавања, брзина је значајно мање битна од квалитета излаза.

9.2.1 Линеарни конгруентни генератор

Ово је без сумње најпопуларнији генератор квази-случајних бројева. Производи секвенцу X_i целих случајних целих бројева користећи формулу

$$X_i = (a \times X_{i-1} + c) \bmod M.$$

Секвенца зависи од избора *seed* вредности X . Уобичајено је да корисник задаје *seed* вредност. У библиотечким имплементацијама, рецимо у стандардној С библиотеци, ова вредност се обично узима из системског сата.

Када је $c = 0$, генератор се назива **мултипликативним конгруентним**. Све три вредности, a , c и M морају бити пажљиво одабране да бисмо се осигурали да секвенца има дуг период и добре особине насумичности. Максимални период је M . За 32-битне целе бројеве, максимални период је 2^{32} , што је нешто више од 4 милијарде. Ово је премало за модерне рачунаре који извршавају милијарде операција у секунди. Квалитетан линеарни конгруентни генератор мора да има барем 48 бита прецизности.

Линеарне конгруентне методе могу да се употребе и да генеришу бројеве са покретним зарезом. Пошто генератор даје целе бројеве у интервалу од 0 до $M - 1$, дељење X_i са M даје број у покретном зарезу који припада интервалу $[0, 1)$.

Најзначајнија мана линеарног конгруентног генератора огледа се у чињеници да су најмање значајни битови повезани, посебно када је M степена 2. Поред тога, уређене k -торке $(X_i, X_{i+1}, \dots, X_{i+k-1})$ најртране у k -димензионој јединичној хиперкоцки формирају решетку. Овај проблем постаје све израженији како се k повећава, па може да утиче на квалитет вишедимензионих симулација које се ослањају на конгруентни генератор случајних бројева.

9.2.2 Лаговани Фибоначи генератор

Лаговани Фибоначи генератор производи секвенце са изузетно дугим периодима, а уз то је и доволно брз. Формула којом се генеришу чланови секвенце је:

$$X_i = X_{i-p} * X_{i-q}.$$

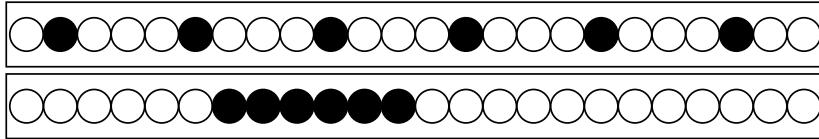
p и q су кашњења, $p > q$, док је $*$ било која бинарна операција. Погодне операције су сабирање по модулу M , одузимање по модулу M , множење по модулу M и битовско ексклузивно ИЛИ. Са сабирањем и одузимањем даје и целе и реалне бројеве. Ако даје реалне, M мора бити 1. Са множењем даје само непарне целе бројеве.

За разлику од линеарног конгруентног генератора којем треба само једна *seed* вредност, лаговани Фибоначи захтева p *seed* вредности, и то X_0, X_1, \dots, X_{p-1} . Пажљиви избор $p, q, M, X_0, \dots, X_{p-1}$ резултира секвенцама са веома дугим периодима и добром насумичношћу. Ако M има b битова (тј. ако X_i имају b битова), максимални период за адитивни лаговани Фибоначи је $(2^p - 1)2^{b-1}$. Повећање лага p повећава захтеве за меморијом, али и повећава максимални период.

9.3 Паралелни генератор случајних бројева

Паралелне Монте-Карло методе зависе од наше способности да генеришемо велики број високо квалитетних секвенци случајних бројева. **Идеални** паралелни генератор случајних бројева има све особине идеалног секвенцијалног генератора, али и још додатно:

1. **Нема корелација** између бројева у различитим секвенцама.
2. **Скалабилност.** Може да се опслужи велики број процеса, сваки са својим током случајних бројева.
3. **Локалност.** Процес треба да може да генерише своју секвенцу без икакве међупроцесне комуникације.



Слика 9.3: Паралелни генератори случајних бројева. *Горе:* Метода жабљег скока. *Доле:* Метода поделе на секвенце. На обе слике означени су елементи који припадају процесу ранга 1 (од укупно 4)

9.3.1 Метода господар-слуга

Процес господар има задатак да генерише случајне бројеве и дистрибуира их слугама које их конзумирају. Алгоритам има две значајне мане. Прва је што одређене врсте генератора узоркују случајне бројеве из исте секвенце, па веома удаљене корелације у оригиналној секвенци могу постати близске корелације у паралелним секвенцима. Друга значајна мана је очигледан недостатак скалабилности.

9.3.2 Метода жабљег скока

Алгоритам жабљег скока је аналогон цикличној алокацији података процесима. Ако претпоставимо да имамо укупно p процеса, процес са рангом r ће узети сваки p -ти случајни број из секвенце:

$$X_r, X_{r+p}, X_{r+2p}, \dots$$

Прилично је једноставно модификовати линеарни конгруентни генератор да имплементира методу жабљег скока. Наиме, код њега треба заменити a са $a^p \bmod M$ и c са $c(a^p - 1)/(a - 1) \bmod M$. Метода је илустрована на Слици 9.3 горе.

Највећи проблем овог генератора је што, за одређене вредности p , елементи новонасталих секвенци могу да буду корелисани, иако елементи оригиналне секвенце нису корелисани. То је посебно вероватно ако је p степен двојке, користи се линеарни конгруентни генератор и M је такође степен двојке. Друга значајна мана, је што овај генератор не подржава динамичко креирање нових секвенци.

9.3.3 Метода поделе на секвенце

Подела на секвенце је аналогон блок декомпозицији података процесима. Претпоставимо да користимо генератор псевдослучајних бројева који има период P . Идеја је да се првих P бројева које емитује генератор се поделе на делове једнаке дужине, по једна за сваки процес. Метода је илустрована на Слици 9.3 доле.

Главна мана је што је сваки процес принуђен да се, при иницијализацији, помери до почетка своје секвенце. Ова активност може да траје дugo, али се, срећом, изводи само једанпут. Линеарни конгруентни генератори са модулом који је степен двојке често резултују корелацијама на дугим интервалима. Како токови различитих процеса репрезентују међусобно доста удаљене елементе, могуће су корелације међу процесима.

9.3.4 Метода параметризације

Четврти (и најбољи) начин за паралелно генерисање случајних бројева је да сваки процес покрене сопствени генератор, уз осигурање да се сваки иницијализује различитим параметрима. Код линеарног конгруентног генератора можемо, на пример, употребити различите адитивне константе да бисмо добили различите секвенце.

Лаговани Фиbonачи генератори су посебно погодни за овакав приступ. Доделом посебне табеле лаг вредности сваком процесу постижемо да сваки процес крене да генерише различиту секвенцу случајних бројева. Наравно, корелације у почетној табели лагова би биле фаталне. Један начин за превазилажење овог проблема је да се искористи неки *други* генератор случајних бројева за попуњавање табеле. Процеси могу да искористе технику жабљег скока или поделе у секвенце, како би попунили своје табеле.

Број различитих токова случајних бројева који на овај начин могу да се добију је веома велики [12]. На пример, подразумевани генератор SPRNG библиотеке има око 2^{1008} различитих токова [12].

9.4 Неуниформне расподеле

У овом одељку ће бити дефинисане неке расподеле које се често користе у реалним проблемима, заједно са конкретним применама.

9.4.1 Инверзна кумулативна функција дистрибуције

Уколико нам је потребно да генеришемо случајне бројеве који подлежу некој расподели која није унiformна, морамо се позабавити терминологијом непрекидних случајних променљивих. Пре свега, уведимо функцију **густине вероватноће** $f(x)$. Густина вероватноће је ненегативна функција дефинисана на скупу реалних бројева \mathbb{R} , таква да је вероватноћа да случајна променљива узме вредност из интервала $[a, b]$ за свако $a \leq b$ дата интегралом:

$$Pr(a \leq x \leq b) = \int_a^b f(x) dx.$$

Наравно, интеграл на целом интервалу дефинисаности даје вероватноћу сигурног догађаја:

$$Pr(a \leq x \leq b) = \int_{-\infty}^{\infty} f(x) dx = 1. \quad (9.1)$$

Други појам који уводимо је **кумулативна функција** или **функција дистрибуције** $F(X)$. На основу познате густине вероватоће показује колика је вероватноћа да случајна променљива x поприми вредност мању или једнаку некој појединачној вредности X :

$$F(X) \equiv F(x \leq X) = \int_{-\infty}^X f(x) dx.$$

На основу адитивних особина одређених интеграла, добијамо везу између вероватноће и кумулативне функције:

$$\begin{aligned} Pr(a \leq x \leq b) &= \int_a^b f(x) dx \\ Pr(a \leq x \leq b) &= \int_{-\infty}^b f(x) dx - \int_{-\infty}^a f(x) dx \\ Pr(a \leq x \leq b) &= F(b) - F(a). \end{aligned}$$

Да би се узорковали одбирачи који подлежу некој одређеној густини расподеле потребно је одредити инверзну функцију кумулативне функције дистрибуције, што ће бити објашњено на конкретном примеру у наредном одељку. Илустрација густине вероватноће и кумулативне функције приказана је на Слици 9.4.

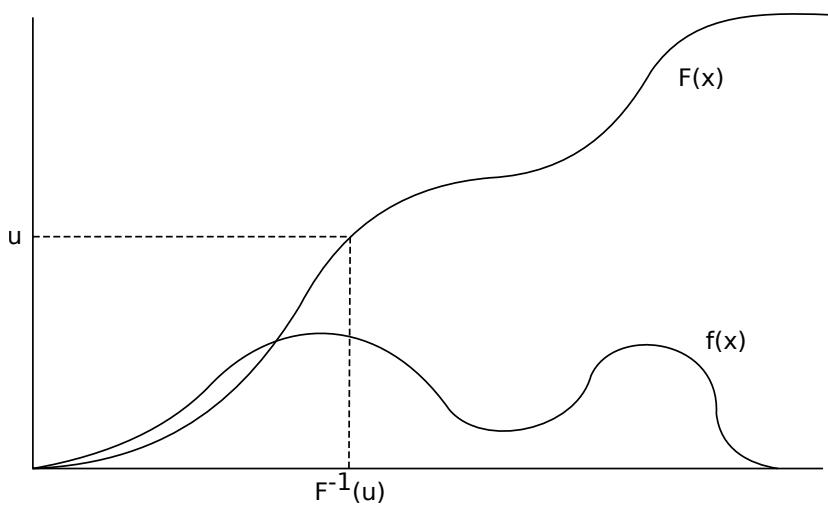
9.4.2 Случајне тачке унутар кружнице

Задатак је, слично као и у примеру са бројем π , генерисати случајне тачке унутар кружнице полупречника R , уз услов да њихова густина буде хомогена. Један приступ је да употребимо исту логику као у примеру са бројем π , с тим што бисмо прихватили само оне случајне тачке унутар круга.

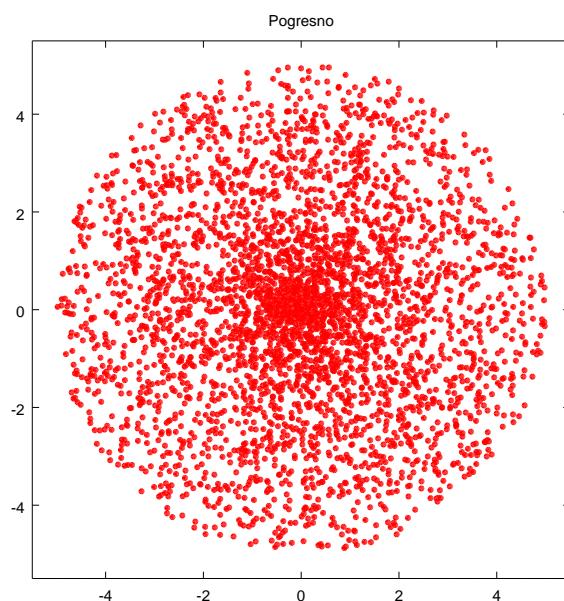
Међутим, овај приступ **није оптималан**, јер смо неке случајне тачке генерирали, а нисмо их искористили. Следећа идеја је коришћење поларних координата. Случајно ћемо бирати $\varphi \in [0, 2\pi]$ и $r \in [0, R]$. Лапчки претпоставимо да то можемо учинити само коришћењем унiformних случајних променљивих u_1 и u_2 , које припадају интервалу $[0, 1]$:

$$\varphi = 2\pi \cdot u_1, \quad (9.2)$$

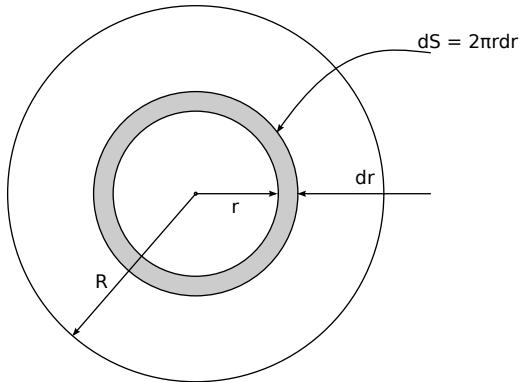
$$r = R \cdot u_2. \quad (9.3)$$



Слика 9.4: Густина расподеле и кумулативна функција расподеле



Слика 9.5: Распоред случајних тачака унутар кружнице добијен коришћењем израза 9.2 и 9.3



Слика 9.6: Инфинитезимални елемент површине круга. Са повећањем r линеарно се повећава и ова површина

Међутим, на овај начин, добићемо гушће распоређене тачке у центру круга него на периферији, као што се види на Слици 9.5. Разлог томе је погрешан приступ генерисању случајних одбираца за r , који очигледно више не може бити униформан.

Ако посматрамо појас инфинитезималне ширине dr (Слика 9.6), који се налази на удаљености r од центра, његову површину можемо изразити као:

$$dS = O \cdot dr = 2\pi r \cdot dr$$

Дакле, површина овог појаса, па самим тим и број тачака, су директно пропорционални r . Неки други појас, ближи периферији круга, би имао већу површину, а самим тим и број случајних тачака које треба генерисати. Закључак ове дискусије је да густина вероватноће мора бити пропорционална r :

$$f(r) = A \cdot r,$$

где је A константа. Како да одредимо константу? Па на основу једначине (9.1):

$$\int_0^R f(r) dr = \int_0^R Ar dr = A \int_0^R r dr = A \cdot \frac{r^2}{2} \Big|_0^R = 1$$

Одатле је $A = \frac{2}{R^2}$, што даје коначан израз за густину вероватноће:

$$f(r) = \frac{2}{R^2} r.$$

Како сада да извучемо случајни број из ове функције густине вероватноће? Кумулативна функција дистрибуције $F(X)$ даје вероватноћу да се изабере било која вредност мања или једнака од X .

$$F(r) = \int_0^r f(r)dr = \int_0^r \frac{2}{R^2} \cdot r dr = \frac{2}{R^2} \int_0^r r dr = \frac{2}{R^2} \cdot \frac{r^2}{2} = \frac{r^2}{R^2} = u_1$$

Пошто је $r = F^{-1}(u_1)$, следи:

$$u_1 = \frac{r^2}{R^2}$$

тј.

$$r = R\sqrt{u_1}. \quad (9.4)$$

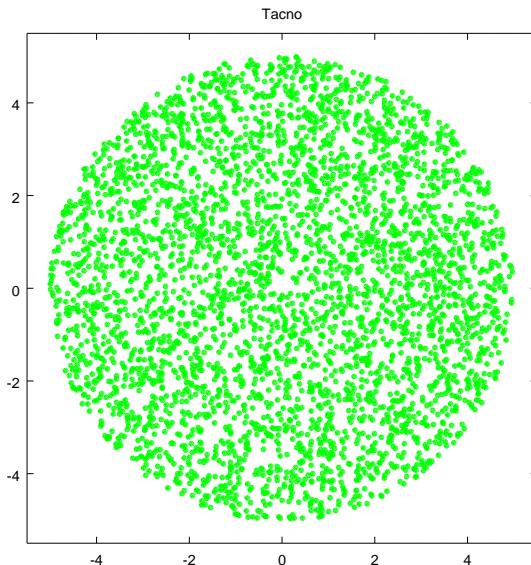
Кумулативна функција дистрибуције може да узме вредности од 0 до 1. Растућа је, и расте брже у оној тачки где је густина већа. Случајан број из дате густине вероватноће је инверзна кумулативна функција дистрибуције. Када је густина вероватноће висока, кумулативна функција брзо расте, па за велики распон бројева u_1 добијамо приближно исте вредности за r . Када је густина мала, кумулативна функција споро расте, па се за малу промену вредности u_1 , вредност r пуно промени. Сада, на основу једначине (9.4) можемо да генеришемо хомоген распоред тачака унутар круга, као на Слици 9.7. Изворни код у програмском језику *Octave* који је коришћен за генерисање слика 9.5 и 9.7 дат је на Листингу 9.1.

Листинг 9.1: Генерисање случајних тачака у кругу

```

1 clear all
2 close all
3
4 Nmax=4000;
5 R=5;
6
7 for n=1:Nmax
8
9     % Pogresan metod
10    r1(n)=R*rand(1,1);
11    theta1(n)=2*pi*rand(1,1);
12
13    % Pravi metod
14    % Gustina verovatnoce pdf_r(r)=(2/R^2) * r
15    % Kumulativna funkcija pdf_r je F_r = (2/R^2)* (r^2)/2
16    % Inverzna kumulativna je r = R*sqrt(F_r)
17    % Tako da se r generise kao
18    r2(n) = R*sqrt(rand(1,1));
19    % a teta kao i ranije
20    theta2(n)=2*pi*rand(1,1);

```



Слика 9.7: Случајне тачке унутар круга генерисане коришћењем израза (9.4)

```

21
22 % Konverzija u dekartove koordinate
23 x1(n)=r1(n)*cos(theta1(n));
24 y1(n)=r1(n)*sin(theta1(n));
25 x2(n)=r2(n)*cos(theta2(n));
26 y2(n)=r2(n)*sin(theta2(n));
27 end
28
29 subplot(1,2,1)
30 plot(x1,y1,'r.')
31 axis([-1.1*R 1.1*R -1.1*R 1.1*R])
32 axis square
33 title('Pogresno')
34 subplot(1,2,2)
35 plot(x2,y2,'g.')
36 axis([-1.1*R 1.1*R -1.1*R 1.1*R])
37 axis square
38 title('Tacno')

```

9.4.3 Експоненцијална расподела

Експоненцијална функција густине вероватноће моделује распад радиоактивних атома, растојање које неутрон пређе кроз препреку пре него што се судари са атомом исл. Како долазимо до експоненцијалног облика? Најједноставније

је посматрати проблем распада неког радиоактивног материјала. Број распаднутих честица је директно сразмеран броју преосталих (нераспаднутих) честица и времену. У диференцијалном запису:

$$dN = -N \frac{1}{m} dt$$

Знак минус фигурише јер се број нераспаднутих честица смањује. Ако ову диференцијалну једначину интегралом методом раздвајања променљивих, добија се:

$$\begin{aligned} \frac{dN}{N} &= -\frac{1}{m} dt \\ \int_{N_0}^N \frac{dN}{N} &= -\frac{1}{m} \int_0^t dt \\ \ln N \Big|_{N_0}^N &= -\frac{1}{m} t \Big|_0^t \\ \ln N - \ln N_0 &= -\frac{1}{m} t \\ \ln \frac{N}{N_0} &= -\frac{1}{m} t \end{aligned}$$

Одатле је закон промене броја честица при радиоактивном распаду:

$$N = N_0 e^{-\frac{1}{m} t}.$$

Дакле, густина вероватноће је сразмерна са $e^{-\frac{1}{m} t}$. Ако применимо закон нормирања (9.1), густина вероватноће имаће облик:

$$f(x) = \frac{1}{m} e^{-\frac{x}{m}},$$

при чему је m тзв. очекивана вредност (математичко очекивање), које се за континуалну расподелу дефинише као:

$$E(x) = \int_0^\infty x \cdot f(x) dx = \frac{1}{m} \int_0^\infty x e^{-\frac{x}{m}} dx = m.$$

Даље, да бисмо добили кумулативну функцију $F(x)$ морамо да интегралимо $f(x)$ користећи смену $t = -x/m$:

$$\begin{aligned} F(x) &= \int_0^x f(x) dx \\ F(x) &= \frac{1}{m} \int_0^x e^{-\frac{x}{m}} dx \\ F(x) &= \frac{1}{m} \int_0^{-\frac{x}{m}} e^t (-mdt) \\ F(x) &= - \int_0^{-\frac{x}{m}} e^t dt \\ F(x) &= -e^t \Big|_0^{-\frac{x}{m}} \\ F(x) &= 1 - e^{-\frac{x}{m}} \end{aligned}$$

Да бисмо правилно узорковали из експоненцијалне расподеле, треба да нађемо инверзну функцију $F^{-1}(u)$. Како је $F^{-1}(F(x)) = x$, односно $F^{-1}(1 - e^{-\frac{x}{m}}) = x$, сменом $1 - e^{-\frac{x}{m}} = t$ добијамо да је $x = -m \cdot \ln(1 - t)$. Коначно је $F^{-1}(t) = -m \cdot \ln(1 - t)$, или:

$$F^{-1}(u) = -m \cdot \ln(1 - u).$$

Пошто је u униформно расподељено између 0 и 1, нема разлике између u и $1 - u$. Одатле је функција

$$F^{-1}(u) = -m \cdot \ln u$$

експоненцијално расподељена са средњом (очекиваном) вредношћу m .

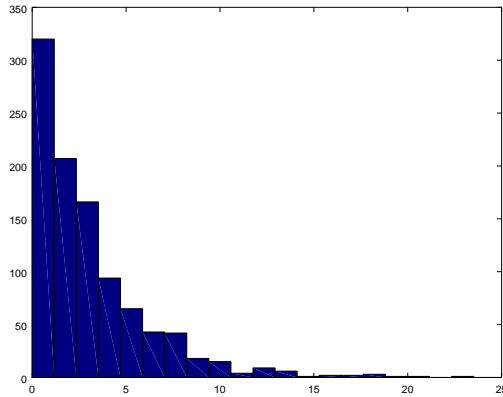
Пример 9.1. Произвести 100 узорака из експоненцијалне дистрибуције са средњом вредношћу 3 и резултате приказати графички у форми хистограма.

Листинг 9.2: Експоненцијална расподела са очекиваном вредношћу 3

```

1 u=rand(500,1);
2 s=-3*log(u);
3 hist(s, 20);
4 mean(s)

```



Слика 9.8: Експоненцијална дистрибуција са очекиваном вредношћу 3

9.4.4 Бокс-Милерова трансформација

Бокс-Милерова трансформација је метода узорковања псеудослучајних бројева која генерише парове независних случајних бројева који подлежу нормалној (Гаусовој) дистрибуцији. Као и у случају експоненцијалне дистрибуције, полази се од низа униформних случајних бројева.

Пођимо од формуле за нормалну дистрибуцију са очекиваном вредношћу нула и јединичном стандардном девијацијом за две случајне променљиве x и y :

$$p(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}, \quad p(y) = \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}}.$$

Њиховим множењем ћемо добити:

$$p(x, y) = \frac{1}{2\pi} e^{-\frac{x^2+y^2}{2}} = \left(\frac{1}{2\pi} \right) \cdot \left(e^{-\frac{r^2}{2}} \right), \quad (9.5)$$

где је:

$$x = r \cdot \cos(\theta) \text{ и } y = r \cdot \sin(\theta) \quad (0 \leq r \leq 1, \theta \in \{0, 2\pi\}).$$

Функција $p(x, y)$ је, према (9.5), производ две густине вероватноће, једне експоненцијалне која зависи од квадрата полупречника и једне униформне која зависи од угла. Генерирање узорака из униформне расподеле је елементарно:

$$\theta = 2\pi \cdot u_1,$$

док за експоненцијалну користимо инверзну кумулативну функцију расподеле:

$$\begin{aligned} u_2 &= e^{-\frac{r^2}{2}} \\ \ln u_2 &= -\frac{r^2}{2} \\ r &= \sqrt{-2 \cdot \ln u_2}. \end{aligned}$$

На основу ових једначина може да се напише и програм дат на Листингу 9.3.

Листинг 9.3: Бокс-Милеров метод

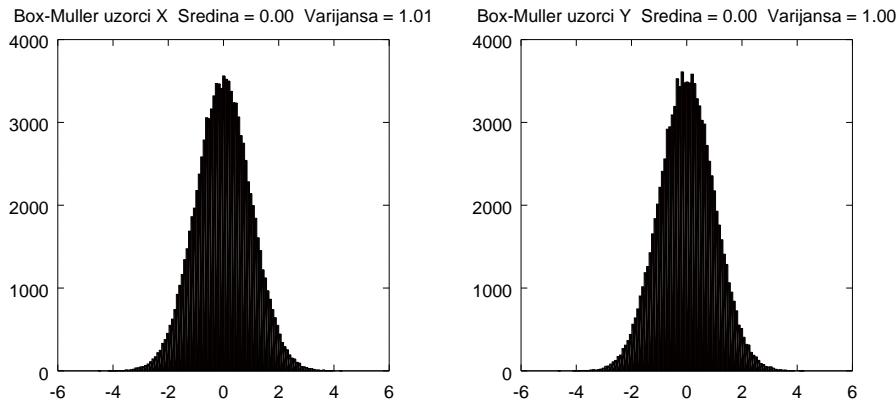
```

1 % Uzorci iz normalne raspodele koriscenjem Box-Muller metode
2 u = rand(2,100000);
3 r = sqrt(-2*log(u(1,:)));
4 theta = 2*pi*u(2,:);
5 x = r.*cos(theta);
6 y = r.*sin(theta);
7
8 % Graficki prikaz
9 figure
10 % X uzorci
11 subplot(121);
12 hist(x,100);
13 colormap hot;axis square
14 title(sprintf('Box-Muller uzorci X\n Sredina = %1.2f\n Varijansa =
    %1.2f',mean(x),var(x)))
15 xlim([-6 6])
16
17 % Y uzorci
18 subplot(122);
19 hist(y,100);
20 colormap hot;axis square
21 title(sprintf('Box-Muller uzorci Y\n Sredina = %1.2f\n Varijansa =
    %1.2f',mean(y),var(y)))
22 xlim([-6 6])

```

9.4.5 Метода одбацивања

Методу одбацивања је први употребио Џон фон Нојман да би узорковао случајне бројеве из густине расподеле коју је тешко или немогуће интегралити аналитички. Претпоставимо да смо у могућности да генеришемо узорке из неке друге функције густине вероватноће $h(x)$, као и да можемо да утврдимо константу такву да је $f(x) \leq \delta h(x)$, за свако x (Слика 9.10). Узорке из $f(x)$ добијамо тако што генеришемо узорак x_i из h и други узорак u_i из униформне дистрибуције.



Слика 9.9: Бокс-Милерова метода за генерирање узорака нормалне расподелe

Ако је $u_i \delta h(x_i) \leq f(x_i)$ онда прихватамо x_i као узорак из $f(x)$ и враћамо га. У супротном, понављамо тест са другим x_i и другим u_i .

Дакле, тачке облика $(x_i, u_i \delta h(x_i))$ униформно узоркују простор испод криве $\delta h(x)$. Пошто прихватамо само тачке испод $f(x)$, резултујућа секвенца случајних бројева рефлектује баш функцију густине вероватноће $f(x)$.

Метода најбоље функционише када је разлика између $f(x)$ и $\delta h(x)$ релативно мала. Што је разлика између ове две криве већа, то је и фреквенција већа, тиме успоравајући процес узорковања. Такође, ефикасност ове методе значајно опада и приликом повећања броја димензија.

Пример 9.2. Дата је функција густине вероватноће облика:

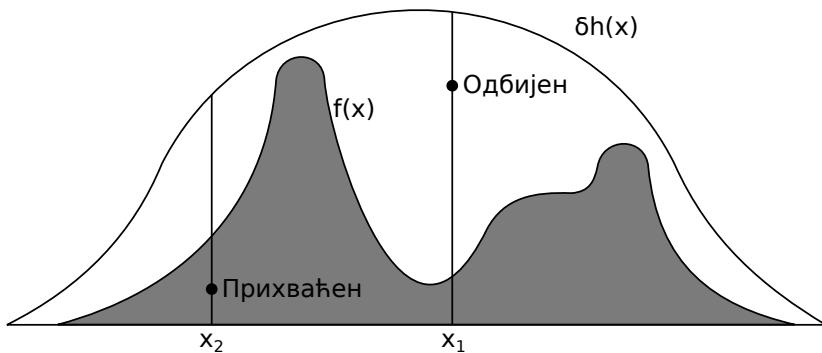
$$f(x) = \begin{cases} \sin(x), & 0 \leq x \leq \pi/4, \\ (-4x + \pi + 8)/(8\sqrt{2}), & \pi/4 < x \leq 2 + \pi/4, \\ 0, & \text{у осталим случајевима} \end{cases}$$

Узорковати случајне вредности из ове густине вероватноће користећи методу одбацивања.

Решење: Треба пронаћи δ и $h(x)$ тако да $(x) \leq \delta h(x)$ за свако x . Примећујемо да је функција густине расподеле различита од нуле за вредности x између 0 и $2 + \pi/4$, као и да је максимална вредност $\sqrt{2}/2$. Бирајмо униформну функцију густине расподеле за $h(x)$:

$$h(x) = \begin{cases} 1/(2 + \pi/4), & 0 \leq x \leq 2 + \pi/4, \\ 0, & \text{у осталим случајевима} \end{cases}$$

Ако сада помножимо $h(x)$ са $\delta = (2 + \pi/4) \cdot (\sqrt{2}/2)$, онда је задовољено да је



Слика 9.10: Метода одбаџивања. Тачке унутар сенченог региона ће бити прихваћене, а тачке ван њега одбачене.

$\delta h(x) \geq f(x)$ за свако x . Упрошћавањем добијамо:

$$\delta h(x) = \begin{cases} \sqrt{2}/2, & 0 \leq x \leq 2 + \pi/4, \\ 0, & у осталим случајевима \end{cases}$$

Решење у програмском језику *Octave* (Matlab) дато је на Листингу 9.4, а резултат извршења скрипте на Слици 9.11.

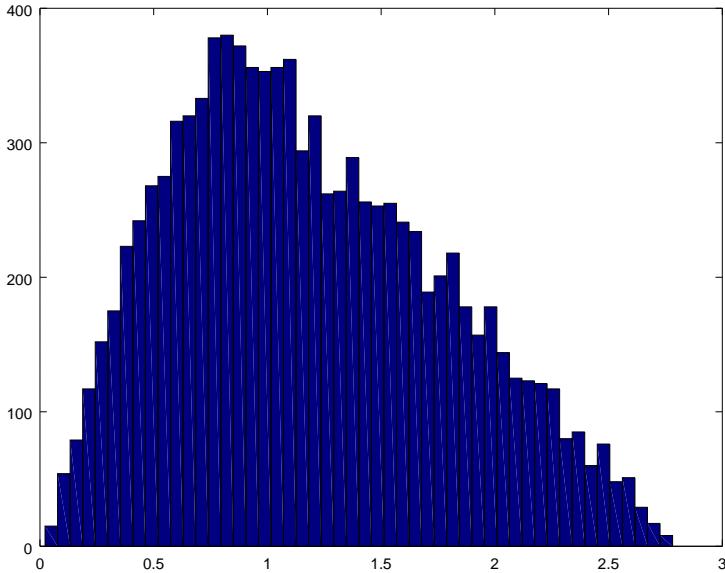
Листинг 9.4: Пример за методу одбаџивања

```

1 %
2 % F-ja koja racuna vrednost distribucije verovatnoce
3 %
4 function vrednost = f(x)
5   if (0<=x) && (x<=pi/4)
6     vrednost = sin(x);
7   elseif (pi/4<=x) && (x<=2+pi/4)
8     vrednost = (-4*x+pi+8) / (8*sqrt(2));
9   else
10    vrednost = 0;
11  end
12 endfunction
13
14 i=1;
15 while i < 10000
16   x = rand*(2+pi/4);
17   u = rand;
18   if u*sqrt(2)/2 <= f(x)
19     y(i) = x;
20     i = i + 1;
21   end
22 end

```

```
23 hist(y, 50);
```



Слика 9.11: Метода одбацивања - пример

9.5 Студије случаја

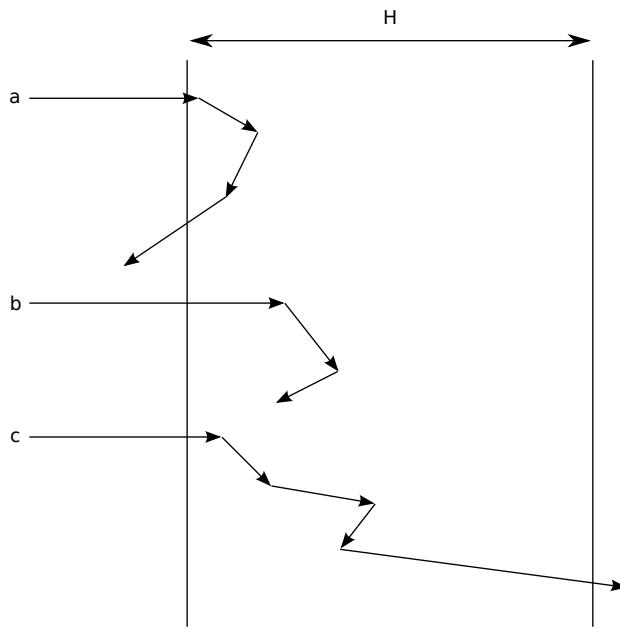
У овом делу ће у најопштијим цртама бити дато неколико конкретних примера велике употребне вредности Монте-Карло метода.

9.5.1 Траспорт неутрона

Разматрамо упрошћени модел транспорта неутрона у две димензије (Слика 9.12). Извор еmitује неутроне на хомогену плочу дебљине H и бесконачне висине. Неутрон може бити рефлектован од стране плоче, може бити апсорбован и може проћи кроз плочу. Наш задатак је да израчунамо проценат оних који ће проћи као функцију дебљине H .

Две константе које описују интеракцију неутрона са атомима плоче су ефикасни пресек за захват неутрона C_c и ефикасни пресек за расејање C_s . Укупни ефикасни пресек интеракције је $C = C_c + C_s$.

Дужина L коју ће пропутовати неутрон у плочи пре него интерагује са следећим атомом моделује се експоненцијалном дистрибуцијом са средњом вредно-



Слика 9.12: Транспорт неутрона

шћу $1/C$. Као што је објашњено у Одељку 9.4.3, то се постиже формулом:

$$L = -\frac{1}{C} \ln u,$$

где је u случајни број из униформне расподеле. Када једном дође до интеракције с атомом, вероватноћа расејања је C_s/C , а вероватноћа захвата је C_c/C . Када се неутрон расеје, подједанком вероватноћом се креће у било ком правцу d између 0 и π (у радијанима). Како је плоча бесконачно висока, није битно да ли се креће наниже или навише. Дистанца коју неутрон пређе у x правцу је $L \cos(d)$. Симулација за дати неутрон се завршава у тренутку када наступи било који од следећа три догађаја:

1. Неутрон бива апсорбован од стране атома.
2. Ако је x позиција мања од нуле, значи да је плоча рефлектовала неутрон.
3. Ако је x позиција већа од H , значи да је плоча пропустила неутрон.

Псеудокод за ову симулацију дат је у Алгоритму 4. Треба приметити да у овом симулацији не фигурише време, већ да је ток диктиран догађајима интеракције неутрона са атомима. Ова, *псеудо-временска* прогресија назива се још и **Монте-Карло време**.

Алгоритам 4 Транспорт неутрона - псевдокод

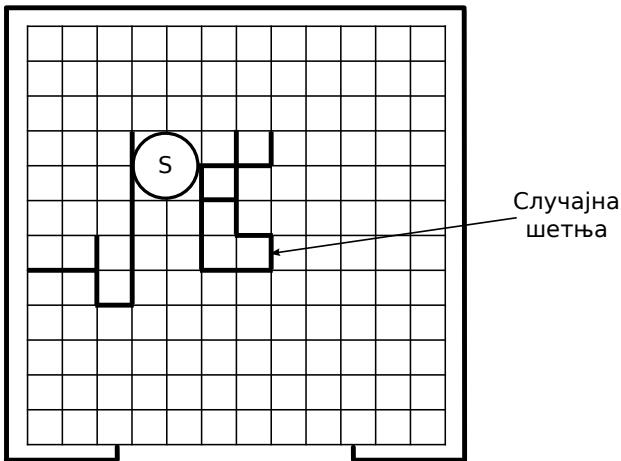
```

1:  $C$  - Средња дистанца између две интеракције је  $1/C$ 
2:  $C_s$  - Компонента расејања
3:  $C_c$  - Компонента захвата
4:  $H$  - Дебљина плоче
5:  $L$  - Дистанца коју неутрон пређе пре колизије
6:  $d$  - Смер кретања неутрона
7:  $u$  - Униформни случајни број
8:  $x$  - Позиција неутрона у плочи ( $0 \leq x < H$ )
9:  $n$  - Број узорака
10:  $a$  - Тачно уколико неутрон још путује у плочи
11:  $r, b, t$  - Број рефлексованих, апсорбованих и трансмитованих неутрона
12: Почетак
13:  $r, b, t \leftarrow 0$ 
14: for  $i \leftarrow 0$  to  $n$  do
15:    $d \leftarrow 0$ 
16:    $x \leftarrow 0$ 
17:    $a \leftarrow \text{true}$ 
18:   while  $a$  do
19:      $L \leftarrow 1/C \cdot \ln u$ 
20:      $x \leftarrow x + L \cdot \cos(d)$ 
21:     if  $x < 0$  then                                ▷ Рефлексован
22:        $r \leftarrow r + 1$ 
23:        $a \leftarrow \text{false}$ 
24:     else if  $x \geq H$  then                  ▷ Трансмитован
25:        $t \leftarrow t + 1$ 
26:        $a \leftarrow \text{false}$ 
27:     else if  $u < C_c/C$  then                ▷ Захваћен
28:        $b \leftarrow b + 1$ 
29:        $a \leftarrow \text{false}$ 
30:     else
31:        $d \leftarrow u \cdot \pi$ 
32:     end if
33:   end while
34: end for
35: Испиши  $r/n, b/n, t/n$ 

```

9.5.2 Температура тачке на дводимензионој плочи

Разматра се врло танка плоча направљена од хомогеног материјала, приказана на Слици 9.13. Желимо да израчунамо температуру стационарног стања у некој тачки на плочи. Врх и дно плоче су изоловани, што значи да температуре било које тачке одређују искључиво тачке које је окружују, осим ако су у питању температуре тачака на границама, које су фиксиране.



Слика 9.13: Одређивање температуре тачке методом случајних шетњи. Граничне тачке које додирују бели оквир имају температуру 0, док оне граничне тачке које додирају сиви оквир имају температуру 100. Означена је једна случајна шетња која полази из тачке S .

Температура стационарног стања је одређена познатом Лапласовом једначином $\Delta^2 T = 0$, што у крајњој линији значи да је температура тачке једнака аритметичкој средини околних тачака. Ако простор дискретизујемо у мрежу (Слика 9.13), Лапласова једначина у дискретном облику (метода коначних разлика) ће гласити:

$$S = \frac{1}{4}(T_s + T_i + T_j + T_z), \quad (9.6)$$

где су T_s , T_i , T_j и T_z температуре северне, источне, јужне и западне тачке, респективно, а S температура посматране тачке. За ову намену можемо користити и Монте-Карло технику узимања случајног суседа n пута, сумирањем температура и дељењем добијене суме са n . Ова техника би дала очекивану вредност (9.6).

Проблем је у томе што у ствари не знамо температуре суседних тачака. Међутим, добра вест је што можемо да користимо исту Монте-Карло технику да бисмо дошли до њихових температура. Рекурзивном применом изложене идеје, чинимо тзв. **случајну шетњу** по плочи. Пошто свака рекурзија мора имати услов за

излазак, и овде је такав случај. Шетња се завршава када се дође до границе која има предефинисану температуру.

Алгоритам ради тако што се креће од S , а затим случајно изабере правац крећања (север, запад, југ, исток). Настављамо да се крећемо по истом принципу док не ударимо у границу. Када се то деси, температура те границе се додаје акумулатору и тако у круг. У свакој итерацији рачунамо средњу вредност граничне температуре свих случајних шетњи које су до тог тренутка предузете. Алгоритам се завршава када средња температура конвергира. Читава замисао делује логично и са лаичке тачке, услед чињенице да ће већи број шетњи завршавати на блиским границама него на оним удаљенијим, па ће и температура бити ближа температури ближе границе.

9.5.3 Проблем доделе соба

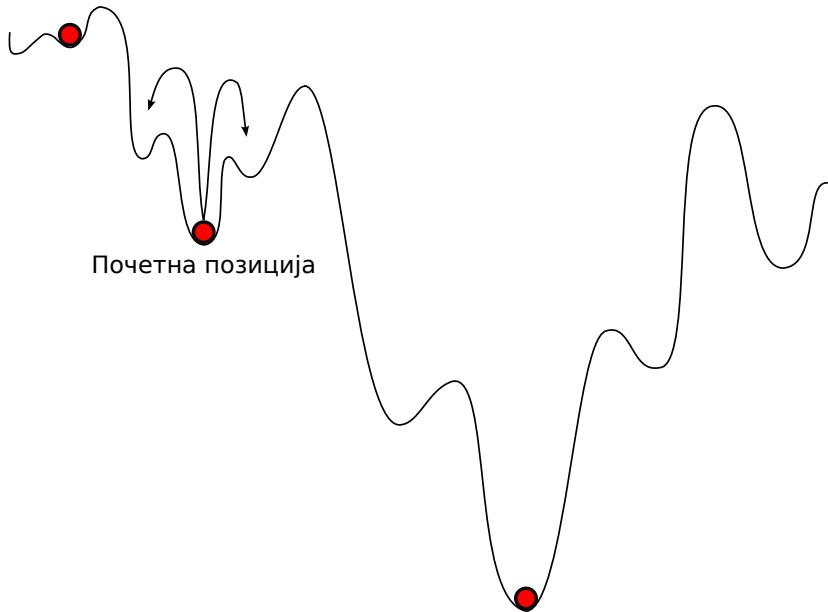
Проблем се састоји у чињеници да треба распоредити n бруцаша у $n/2$ соба у дому, али тако да међусобни конфликти буду минимизовани. Сваки бруцаш је попунио анкету, а компјутерски програм је генерирао матрицу у којој вредност сваког члана $[i, j]$ означава степен неслагања између бруцаша i и бруцаша j . Матрица је симетрична, тј. члан $[j, i]$ једнак је члану $[i, j]$. Овај проблем ћемо решити методом **симулираног каљења**.

Каљење је процес обраде метала у коме се он топи, а затим полако хлади. Када је метал на високој температури, атоми кристалне решетке су покретљивији и лакше се премештају. Како температура опада, кретање је све слабије и атоми метала проналазе своја места, тежећи минимуму енергије (равнотежном стању), које одговара кристалној структури. **Симулирано каљење** за оптимизацију користи аналогију обраде метала каљењем. Решење оптимизационог проблема одговара таквом стању система у коме је одговарајућа циљна функција минимална.

Симулирано каљење је итеративни алгоритам. У свакој итерацији се решење мења на случајан начин, како би се креирало алтернативно решење у околини посјећег. Ако је вредност циљне функције новог решења мања од вредности циљне функције тренутног решења, прелази се на алтернативно решење. Међутим, чак и ако је алтернативно решење лошије од тренутног, постоји вероватноћа, $e^{-\frac{\Delta}{T}}$, да се пређе на алтернативно решење. Δ је разлика вредности циљне функције између два решења, а T тренутна температура.

Зашто бисмо уопште желели да пређемо на лошије решење од онога које већ имамо? Одговор на ово питање очигледан је са Слике 9.14 - не желимо тако брзо да уђемо у локални минимум. Ако је температура виша, већа је вероватноћа да се пређе на решење са лошијом вредношћу циљне функције. Дакле, да бисмо конкретизовали алгоритам симулираног каљења, морамо:

1. Одлучити како да репрезентујемо решења.



Слика 9.14: Алгоритам симулираног каљења. Прелазак на лошије решење могућ је са одговарајућом вероватноћом, која током извршавања алгоритма опада.

2. Дефинисати циљну функцију.
3. Дефинисати начин генерисања нових решења у околини постојећег.
4. Дефинисати функцију хлађења.

Сада ћемо проћи кроз сваки од четири корака за конкретан проблем расподеле соба бруцошима у дому. Полазимо од матрице D , у којој елемент d_{ij} представља степен неслагања бруцоша i са колегом j . Решење проблема је један начин доделе $n/2$ соба за n студената. Свако a_i је цео број између 0 и $n/2 - 1$ и репрезентује собу која је додељена бруцошу i . Свака соба се у овом низу појављује тачно два пута.

Следеће питање је како дефинисати циљну функцију? Циљна функција се може дефинисати као степен општег неслагања у целом дому:

$$\sum_{i=0}^{n-1} d_{i,r_i},$$

где је r_i цимер бруцоша i .

Ново решење у околини посоеђег креирамо када два случајно одабрана студента замене собе.

На крају, али не најмање важно, треба да дефинишемо функцију хлађења. Избор ове функције има велики утицај на перформансе алгоритма, у смислу квалитета добијеног решења и брзине долажења до решења. За овај проблем бирали смо једноставну геометријску прогресију:

$$T_0 = 1$$

$$T_{i+1} = 0,999 \cdot T_i$$

Слика 9.15 илуструје конвергирање решењу са различитим избором почетне вредности температуре.



Слика 9.15: Конвергенција симулираног каљења за две различите вредности почетне температуре

На жалост, симулирано каљење, као и већина тзв. мета-хеуристичких метода оптимизације, не гарантује оптимално решење. Различити токови случајних бројева ће вероватно дати различита решења, иако је алгоритам исти. Из овог разлога је употреба паралелног рачунарства очигледан пут у циљу умањења цеукупног времена рачунања. На Алгоритму 5, дат је псеудо-код решења проблема доделе соба.

Алгоритам 5 Симулирано каљење - псеудокод

1: $a[0..n - 1]$ - низ који садржи распоред по собама
 2: $c1, c2$ - бруцоши укључени у потенцијалну замену соба
 3: $d[0..n - 1, 0..n - 1]$ - улазна матрица која садржи степене неслагања
 4: sum - сума неслагања тренутног решења
 5: new_sum - сума неслагања новог решења
 6: t - температура
 7: **Почетак**
 8: Направи случајан распоред студената по собама
 9: $sum \leftarrow 0$
 10: **for** $i \leftarrow 0$ to $n - 1$ **do**
 11: **for** $j \leftarrow 0$ to $n - 1$ **do**
 12: **if** $a[i] = a[j]$ **then**
 13: $sum \leftarrow sum + d[i][j]$
 14: **end if**
 15: **end for**
 16: **end for**
 17: $t \leftarrow 1$
 18: $i \leftarrow 0$
 19: **while** $i < 1000$ **do** ▷ Заустави ако нема промене за 1000 итерација
 20: **repeat**
 21: $c1 \leftarrow \lfloor u \cdot n \rfloor$
 22: $c2 \leftarrow \lfloor u \cdot n \rfloor$
 23: **until** $a[c1] \neq a[c2]$
 24: Израчунај new_sum под претпоставком да $c1$ и $c2$ замене собе
 25: **if** $new_sum < sum$ **or** $u < e^{(sum-new_sum)/t}$ **then**
 26: Замени собе за $c1$ и $c2$
 27: $sum \leftarrow new_sum$
 28: $i \leftarrow 0$
 29: **else**
 30: $i \leftarrow i + 1$
 31: **end if**
 32: **end while**
 33: Штампај a и sum
 34: **Крај**

Додаци

Додатак I

Big Data приступ

Деценијама је развој алгоритама у области рачунарства високих перформанси био примарно оријентисан на послове симулација и анализе података на највећим могућим скалама. Ова област, иако веома важна, важи за уско специјализовану вештину. Међутим, негде око 2005. године појавила су се два горућа проблема - анализа података на Интернет скали и поплава података из области генетике. Овакви захтеви су звучали сасвим познато људима који се баве рачунарством високих перформанси, али са додатком аудиторијума који чија величина превазилази све са чиме су се до сада сретали.

Одједном су теме као што су скалабилност, поузданост, дистрибуирана линеарна алгебра и смештај огромних количина података постале веома актуелне и чак ургентне. Показало се да се проблеми као што су анализа генома и машинско учење на Интернет скали траже неку другу врсту алата, тј. да за ову сврху MPI са својим приступом ниског нивоа и није најпогоднији. Док се код MPI-а комуникација углавном задржава на транспортном слоју, нови оквири нуде нешто виши ниво апстракције за развој апликација. Алати који доминирају новом, тзв. *Big Data* сценом су:

- **Apache Hadoop** - Дистрибуирани фајл систем и механизам за мапирање и редукцију.
- **Apache Spark** - Оквир за кластер рачунарство са имплицитним паралелизмом података и отпорношћу на грешке [15].
- **Apache HBase** - Омогућава формирање огромних табела података које могу бити дистрибуирани међу чворовима кластера, па се послови мапирања-/редукције могу обављати локално.
- **Apache Hive** - Помоћу језика налик на SQL омогућава манипулацију подацима који се налазе у **HBase** табелама.
- ...

У даљем тексту у оквиру овог поглавља покушаћемо да, помоћу алата **Apache Spark**, решимо неке од проблема који су у претходним поглављима решавани помоћу чистог MPI приступа ниског нивоа. На овај начин показаћемо да алати који су настали у неком сасвим другом миљеу могу бити од велике користи за рачунарство високих перформанси, а при томе и, често за класу погоднији за употребу.

I.1 Apache Spark оквир

Apache Spark је општи софтверски оквир отвореног кода намењен кластер-рачунарству. Омогућава програмирање високог нивоа у језицима као што су *Java*, *Scala*, *Python* и *R*. У пакету се такође налазе и библиотеке вишег нивоа за машинско учење и процесирање графова.

На највишем нивоу, Спартк апликација се састоји од **драјверског програма** који покреће корисничку `main` функцију и извршава паралелне операције на кластеру. Основна апстракција коју Спартк уводи је тзв. *resilient distributed dataset (RDD)*, који представља колекцију елемената који су дистрибуирани на чвровима кластера и којима може да се оперише паралелно. RDD-ови се креирају од фајла (фајлова) на дистрибуираном или локалном фајл систему, постојеће колекције у тзв. драјверском програму исл. Две главне повољне особине RDD структуре је да могу да буду перзистентне у меморији и да се опораве од отказа на компјутационом чвиру.

Друга основна апстракција у Спартку је **дељена променљива** која своју употребну вредност налази у паралелним операцијама. Подразумевано, када Спартк извршава паралелну функцију као скуп задатака на различитим чвровима кластера, он копира сваку променљиву за сваки појединачни задатак. Понекад је, као што смо видели у више наврата у овој књизи, променљиву потребно делити између задатака или између задатака и драјверског програма. Спартк подржава два типа дељених променљивих, **емисионе променљиве**, које се користе за кеширање вредности у меморији свих чвирова и **акумулатори**, који омогућавају редукционе операције, као нпр. бројање и сумирање.

RDD подржава два типа операција: **трансформације**, које креирају нови скуп података од постојећег и **акције**, које враћају неку вредност драјверском програму који је акцију позвао. На пример, `map` је трансформација, јер пролази кроз сваки елемент скupa података и враћа нови, трансформисани RDD. Са друге стране `reduce` је пример акције која агрегира скуп података и враћа вредност драјверском програму (иако постоји и паралелна операција `reduceByKey` која враћа дистрибуирани скуп података).

Све трансформације у Спартку су *лење*, што значи да се не спроводе одмах када су задате. Уместо тога, трансформације које се задају се памте, а рачунају се тек онда када акција захтева повраћај резултата у драјверски програм. Ова особина

омогућава Спарку већу ефикасност. На пример, понекад и није потребно да се велики скуп података добијен мапирањем враћа драјверском програму, него се враћа само релативно мали резултат редукције.

Подразумевано, сваки трансформисани RDD може да се поново прерачуна сваки пут када се над њим предузме нека од акција. Међутим, такође је могуће и кеширати RDD објекат у меморији (перзистенција), када ће Спарк сачувати податке објекта на кластеру у сврху далеко бржег каснијег приступа.

Верзија која је актуелна у време писања овог текста је 2.0.0 и може се преузети са адресе <http://spark.apache.org/downloads.html>. Актуелна верзија ради на било ком UNIX-оликом систему, као што су Линукс и OS-X, где постоји *Java* најмање верзије 7. Након распаковавања одговарајуће архиве, потребно је поставити и неке променљиве окружења. Пошто ћемо примере дати у програмском језику *Python*, ево како се подешава окружење за рад у самосталном начину рада (који је довољан да се испробају приложени примери):

```
$ export PATH=$PATH:.../spark-2.0.0-bin-hadoop2.7/bin
$ export PYSPARK_PYTHON=/usr/bin/python3
$ export PYSPARK_DRIVER_PYTHON=/usr/bin/python3
```

Након тога можемо покренути `pyspark` и радити интерактивно или послати већ спремљени посао Спарку на извршавање. Ево примера и за једно и за друго:

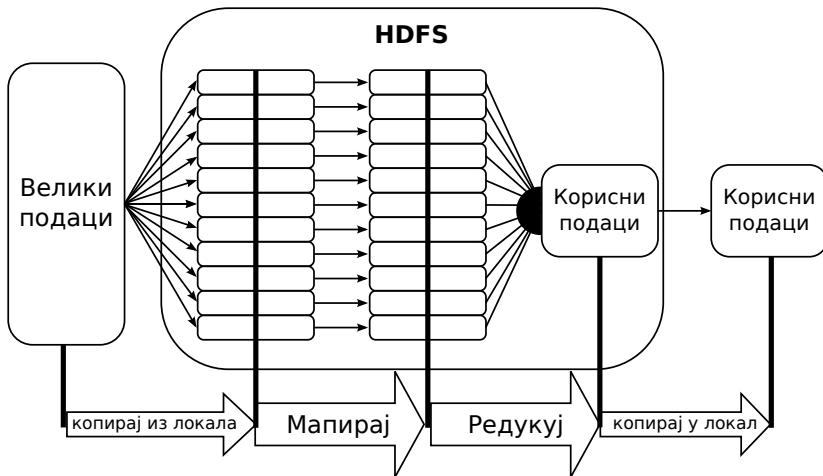
```
$ pyspark --master local[4]      # Spark u lokalu sa 4 procesa
$ spark-submit primer.py        # Slanje posla na izvršenje
```

Дистрибуирани RDD објекти се једноставно добијају из постојећих колекција или фајлова на фајл систему на следећи начин:

Листинг I.1: Креирање RDD објекта

```
1 >>> niz = [1, 2, 3, 4, 5]
2 >>> distNiz = sc.parallelize(niz)
3 >>> linije = sc.textFile("podaci.txt")
4 >>> duzineLinija = linije.map(lambda s: len(s))
5 >>> ukupnaDuzina = duzineLinija.reduce(lambda a, b: a + b)
```

На Листингу I.1 се виде основне операције за које је предвиђен Спарк оквир. Већ у другој линији се позива аутоматски креирана променљива `sc` (*Spark Context*) у сврху дистрибуирања послатог објекта на чворове. У трећој линији се то исто чини и са фајлом са фајл-система, док линије 4 и 5 показују како се једном командом могу извршити две веома честе операције у паралелном програмирању: **мапирање и редукција**. Смисао мапирања је да се за сваку линију врати њена дужина, а редукције да се, на основу ламбда израза у загради, добије укупна дужина фајла у карактерима. Општи принцип функционисања парадигме мапирање/редукција приказан је на Слици I.1.



Слика I.1: Модел мапирање/редукција

I.2 Приближно рачунање броја π

Задатак дефинисан у Одељку 9.1.1 покушаћемо да решимо искључиво коришћењем механизама за мапирање и редукцију. За овај једноставан пример нема улазно-излазних операција. Решење се налази на Листингу I.2.

Листинг I.2: Рачунање броја π Монте-Карло методом

```

1 from random import random
2 from pyspark import SparkContext
3
4 NUM_SAMPLES=10000000
5
6 def primerak(p):
7     x, y = random(), random()
8     return 1 if x*x + y*y < 1 else 0
9
10 sc = SparkContext(appName="Pi primer")
11 count = sc.parallelize(range(NUM_SAMPLES)) \
12     .map(primerak) \
13     .reduce(lambda a, b: a + b)
14
15 print("*****")
16 print ("Pi je priblizno %.15f" % (4.0 * count / NUM_SAMPLES))
17 print("*****")
    
```

Дакле, на мапирању отпадају они примерци који се не налазе унутар јединичног круга (сетују се на нулу), а при редукцији се само изврши сумирање.

I.3 Провођење топлоте методом коначних разлика

Задатак дефинисан у Одељку 3.4 покушаћемо да решимо коришћењем механизама за мапирање и редукцију. Решење се налази на Листингу I.3.

Листинг I.3: Провођење топлоте методом коначних разлика

```

1 import numpy as np
2 from pyspark import SparkContext
3
4 def simulation(sc, ncells, nsteps, nprocs, leftX=-10., rightX=+10.,
5               ao=1.0, coeff=0.375):
6     # Prostorni korak
7     dx = (rightX-leftX)/(ncells-1)
8
9     # Pocetna temperatura stapa
10    def tempFromIdx(i):
11        x = leftX + dx*i
12        return (i, ao*np.sin(np.pi*x))
13
14    # Da li je u putanju tacka u unutrasnjosti?
15    def interior(ix):
16        return (ix[0] > 0) and (ix[0] < ncells-1)
17
18    # Vremenski korak
19    def stencil(item):
20        i,t = item
21        vals = [ (i,t) ]
22        cvals = [ (i, -2*coeff*t), (i-1, coeff*t), (i+1, coeff*t) ]
23        return vals + list(filter(interior, cvals))
24
25    # Glavna rutina simulacije
26    #
27    # Zadavanje pocetne temperature
28    temp = map(tempFromIdx, range(ncells))
29    # Paralelna kolekcija
30    data = sc.parallelize(temp).partitionBy(nprocs)
31    print ("Pocetni uslovi: ")
32    print (data.collect())
33    # Vremenski koraci
34    for step in range(nsteps):
35        stencilParts = data.flatMap(stencil)
36        # Redukuj po kljucu (tacki u prostoru)
37        data = stencilParts.reduceByKey(lambda x,y:x+y)
38        # Odstampaj korak i vrednost temperature na sredini stapa

```

```

39     print("Korak %d maksimalna temperatura %f" % (step, data.
40         lookup(ncells/2) [0]))
41     print ("Konacno: ")
42     print (data.collect())
43 if __name__ == "__main__":
44     sc = SparkContext(appName="Spark Toplotra")
45     simulation(sc, 100, 50, 4, ao=100, leftX=0, rightX=1)

```

Идеја је да се користи иста једначина 3.1 за прираштај температуре у чвору али написана у форми:

$$u_{i,j+1} = u_{i,j} + (R \cdot u_{i-1,j} - 2R \cdot u_{i,j} + R \cdot u_{i+1,j}). \quad (\text{I.1})$$

Ако редни број чвора посматрамо као кључ, а температуру као вредност у колекцији типа речника (мапе), онда се веза између елемената колекције може изразити онако како је то учињено линијама 20 и 21, при чему `cvals` представља прираштаје температуре из једначине I.1. Елементи ове колекције сачињавају директни ациклични граф, који кораком редукције на суму по кључу (линија 38) доводи до коначних вредности температуре у сваком чвору после задатог броја временских корака. Ту је и још неколико интересантних детаља, као што је обезбеђивање очувања граничних услова филтерисањем по критеријуму да ли је у питању чвор који припада унутрашњости (линија 22).

У поређењу са решењем алгоритма коначних разлика које би било имплементирано коришћењем MPI-а, ово решење делује и нешто елегантније.

I.4 Класификација докумената

Задатак дефинисан у Поглављу 8 покушаћемо да рашичланимо и један његов део решимо коришћењем Спарк филтрирања, мапирања и редукције. Решење за један фајл се налази на Листингу I.4.

Листинг I.4: Класификација докумената

```

1 from pyspark import SparkContext
2
3 # Ucitavanje recnika
4 def ucitaj_reci(recnik_fajl):
5     return [rec for line in open(recnik_fajl, 'r') for rec in line.
6             split()]
7
8 # Ucitaj recnik kao listu
9 recnik = ucitaj_reci('recnik.txt')
10
11 sc = SparkContext(appName="Spark klasifikacija")
12 # Emituj recnik svim procesima

```

```
13 recnik_bc = sc.broadcast(recnik)
14
15 # Otvorи fajl ciji se vektor pravi
16 tekst_fajl = sc.textFile("/home/milos/install/spark-2.0.0-bin-
    hadoop2.7/README.md")
17 brojac = tekst_fajl.flatMap(lambda linija: linija.split()) \
18                 .filter(lambda rec: rec in recnik_bc.value) \
19                 .map(lambda rec: (rec, 1)) \
20                 .reduceByKey(lambda a, b: a + b)
21
22 # Odstampaj vektor dokumenta
23 print(brojac.collect())
```

Ово решење је различито од оног понуђеног у Поглављу 8. Иако се дави само једним фајлом, а не свим фајловима из неког директоријума, процесирање тог једног фајла се обавља дистрибуирано захваљујући RDD парадигми и речнику који је објављен свим чворовима у кластеру. Релативно је лако модификовати ово решење да ради над свим фајловима у неком директоријуму. Елементарно решење подразумева (асинхроне) позиве spark-submit за сваки фајл од интеса.

Додатак II

Елементи MPI-2 стандарда

II.1 Стандардизација MPI-а

Као што је више пута напоменуто, MPI (*Message-Passing Interface*) је стандард за креирање паралелних програма. MPI је развијан у две фазе, од стране произвођача паралелних рачунара, писаца библиотека и програмера апликација. Резултат прве фазе развоја, 1993-1994, је прва верзија MPI стандарда, названа MPI-1. Један број важних тема у паралелном рачунарству је намерно изостављен из MPI-1, како би се убрзao излазак нове верзије стандарда. MPI форум се поново састао 1995. да би се размотриле ове теме, као и реализација мањих исправки и појашњења која је захтевао стандард MPI-1. Верзија стандарда MPI-2 објављена је у лето 1997. године.

У одељцима који следе је описан стандардни метод за покретање MPI програма, а затим паралелне улазно/излазне операције у MPI-2 стандарду. Опис могућности да, користећи MPI-2 функције, процес директно приступа подацима другог процеса закључује ово поглавље.

II.2 Портабилни процес покретања

Мали, али веома користан додатак MPI-2 стандарда је стандардни метод за покретање MPI програма, који у ранијем стандарду није био специфициран. Најједноставнији пример позива овог метода је:

```
mpirun -n 16 myprog
```

за покретање програма *myprog* на нпр. 16 процеса. MPI спецификација не говори стриктно како се покреће MPI програм, већ искључиво о механизмима комуникације међу процесима. Од MPI програма се захтева покретање на широком спектру окружења, различитим оперативним системима, менаџерима процеса

итд. Све ово доводи до чињенице да је општи механизам за мулти-процесно покретање веома тешко имплементирати, па је зато изостављен из стандарда.

Међутим, оно што корисници очекују је да програме са једне машине покрену на другој машини без икаквих додатних подешавања. Неколико савремених MPI имплементација, попут OpenMPI [17], користи `mpirun` за покретање MPI програма. Команда `mpirun` се разликује од имплементације до имплементације и захтева различите аргументе. То доводи до конфузије, поготово када су различите MPI имплементације инсталиране на истој машини. Како би прекинули све недоумице, MPI форум је одлучио да се позабави и овим проблемом у верзији стандарда MPI-2. Она препоручује да `mpieexec` буде једини програм за покретање MPI апликација и да су аргументи овог програма тачно одређени и јединствени. Команда

```
mpieexec -n 32 myprog
```

покреће 32 MPI процеса, где је величина MPI_COMM_WORLD комуникатора 32. Назив `mpieexec` је изабран да би се избегли сукоби са различитим варијантама програма `mpirun`.

Поред аргумента `-n <numprocs>`, `mpieexec` има и неколико аргумента који су одређени MPI стандардом. У сваком случају формат за аргументе је `-<назив> вредност`. Неки од стандардних аргумента су:

- *soft*
- *host*
- *arch*
- *wdir*
- *path*
- *file*

```
mpieexec -n 32 -soft 16 prog
```

Значи да, уколико се због ограничења распоређивања процеса, програм не може покренути на 32 процеса, онда да се покрене на 16 процеса.

```
mpieexec -n 4 -host mmilos -wdir /home/milos/izlaz prog
```

Горња команда покреће четири процеса на машини под називом `mmilos` и при том поставља радни директоријум на `/home/milos/izlaz`.

```
mpieexec -n 12 -soft 1:12 -arch sparc-solaris \
-path /home/milos/sunprogs prog
```

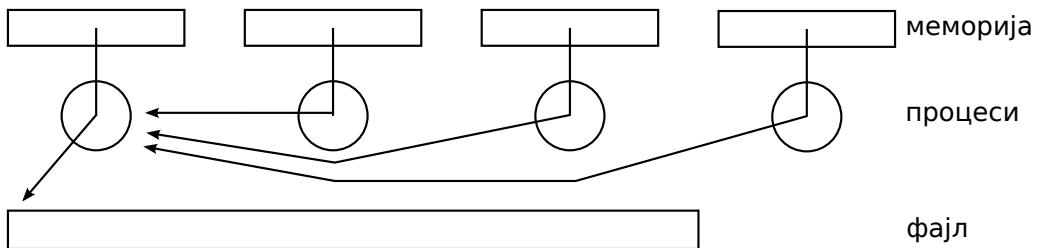
Претходна команда значи да уколико се програм не може покренути на 12 процеса, тада га треба покренути на било ком броју процеса од 1 до 12, на sparc-solaris архитектури, с тим што се програм `prog` налази у наведеном директоријуму.

```
mpieexec -file mojfaajl
```

Значи да `mpieexec` погледа `mojfaajl` за следеће инструкције. Формат тог фајла зависи од конкретне MPI имплементације.

II.3 Паралелне улазно/излазне операције

Паралелне улазно/излазне операције у MPI програму се обављају функцијама које су сличне стандардним улазно/излазним функцијама језика С. MPI омогућава неколико додатних функција које побољшавају учинак и портабилност програма. Основна карактеристика MPI-2 је да може оствари паралелизам у овим операцијама. Секвенцијалне улазно/излазне операције паралелног програма приказане су на Слици II.1.



Слика II.1: Секвенцијалне улазно/излазне операције паралелног програма

Дакле, делегира се један једини процес за улазно/излазни саобраћај, а сви остали процеси се обраћају њему када је потребно читати или уписивати на уређај масовне меморије.

II.3.1 Пример непаралелних У/И операција

Велики део MPI апликација мора да подесује и У/И део, иако MPI-1 стандард не дефинише паралелне У/И операције. Ти делови програма су писани ослањајући се на карактеристике које пружа оперативни систем, најчешће UNIX. Најједноставније је имати један процес који извршава све У/И операције, док други процеси извршавају операције са учитаним подацима. Ако се за пример узме упис низа бројева дужине 100 у фајл, онда дужина података коју обрађује сваки процес зависи од укупног броја процеса. Програм почиње иницијализацијом дела низа за сваки процес. Сви процеси осим процеса са рангом 0 шаљу своје

делове низа процесу 0. Процес 0 уписује свој део низа у фајл, а затим прихвата делове низа од других процеса. Ранг сваког процеса је одређен у функцији MPI_Recv(), тако да се зна редослед пристизања делова низа. Ово је најчешћи начин да се непаралелне У/И операције обаве у паралелном програму који је развијен на основу већ постојећег секвенцијалног програма.

Уколико је *numprocs* = 1, онда нема MPI комуникације. Неки од разлога зашто се У/И операције пишу на овај начин су:

- Паралелни рачунари на којима је покренут програм можда подржавају У/И операције само са једног процеса.
- Могу се користити софистициране У/И библиотеке које су можда писане као део високог нивоа слоја за управљање податцима, а које не подржавају паралелне У/И операције.
- Резултујући фајл је погодан за руковање изван програма (нпр. mv, cp, или ftp).

Учинак програма може бити побољшан омогућавањем процеса да складишти велики блок података. Уколико процес 0 има довољан бафер за податке, он може акумулирати податке других процеса у једниствени бафер за једну велику операцију писања (Листинг II.1). Разлог због кога не треба писати У/И операције на овај начин је недостатак паралелизма, који ограничава учинак и скалабилност, нарочито ако основни фајл систем омогућава паралелне У/И операције.

Листинг II.1: Пример непаралелног У/И програма

```

1 #include <mpi.h>
2 #include <stdio.h>
3 #define BUFSIZE 100
4 int main(int argc, char *argv[])
5 {
6     int i, myrank, numprocs, buf[BUFSIZE];
7     MPI_Status status;
8     FILE *myfile;
9     MPI_Init(&argc, &argv);
10    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
11    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
12
13    for (i=0; i<BUFSIZE; i++)
14    {
15        buf[i] = myrank * BUFSIZE + i;
16    }
17    if (myrank != 0)
18    {
19        MPI_Send(buf, BUFSIZE, MPI_INT, 0, 99, MPI_COMM_WORLD);
20    }
21    else

```

```

22  {
23      myfile = fopen();
24      fwrite(buf, sizeof(int), BUFSIZE, myfile);
25      for (i=1; i<numprocs; i++)
26      {
27          MPI_Recv(buf, BUFSIZE, MPI_INT, i, 99, MPI_COMM_WORLD, &status
28                  );
29          fwrite(buf, sizeof(int), BUFSIZE, myfile);
30      }
31      fclose(myfile);
32  }
33  MPI_Finalize();
34  return 0;
}

```

II.3.2 Пример без улазно/излазних MPI операција

У циљу решавања овог недостатка, следећи корак у миграцији секвенцијалног програма ка паралелном је да се за сваки процес оперише са посебним фајлом, што омогућава паралелни пренос података (Листинг II.2). Овде су У/И операције сваког процеса потпуно независне од У/И операција других процеса. Тако је сваки програм секвенцијални у односу на У/И операције. Наиме, процес отвара свој фајл, уписује податке у њега, а затим га и затвара. Најбоље је да се у називу излазног фајла налази и ранг процеса. Предност овог приступа је да се У/И операције могу одвијати паралелно, а и даље се могу користити секвенцијалне У/И библиотеке. Основни недостатак оваквог приступа је принудно баратање са више фајлова уместо само једног. Поред тога, недостаци овакве шеме су:

- Фајлови се морају спојити пре него што буду коришћени као улаз у другом програму.
- Може се десити да програм који чита ове фајлове мора бити паралелни и покренут са истим бројем процеса.
- Тешко је држати скуп фајлова као групу ради копирања, премештања и слања путем мреже.

Листинг II.2: Програм без улазно/излазних MPI операција

```

1 #include <mpi.h>
2 #include <stdio.h>
3 #define BUFSIZE 100
4 int main(int argc, char *argv[])
5 {
6     int i, myrank, buf[BUFSIZE];
7     char filename[128];

```

```

8   FILE *myfile;
9   MPI_Init(&argc, &argv);
10  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
11  for (i=0; i<BUFSIZE; i++)
12  {
13    buf[i] = myrank * BUFSIZE + i;
14  }
15  sprintf(filename, , myrank);
16  myfile = fopen(filename, );
17  fwrite(buf, sizeof(int), BUFSIZE, myfile);
18  fclose(myfile);
19  MPI_Finalize();
20  return 0;
21 }

```

Такође, ефикасност може да буде умањена уколико смо покренули велики број процеса. То ће довести до великог броја У/И операција са малим бројем података.

II.3.3 MPI У/И операције са одвојеним фајловима

MPI У/И програм је сличан као и претходни програм, с тим што се све У/И операције извршавају MPI функцијама (Листинг II.3). Овакав програм има неколико предности и неколико мана. Прва разлика је што је декларација FILE замењена са MPI_File као типом myfile. Сада је променљива *myfile* типа MPI_File, уместо показивач на објекат типа FILE.

Листинг II.3: MPI програм са одвојеним фајловима

```

1 #include <mpi.h>
2 #include <stdio.h>
3 #define BUFSIZE 100
4 int main(int argc, char *argv[])
5 {
6   int i, myrank, buf[BUFSIZE];
7   char filename[128];
8   MPI_File myfile;
9   MPI_Init(&argc, &argv);
10  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
11  for (i=0; i<BUFSIZE; i++)
12  {
13    buf[i] = myrank * BUFSIZE + i;
14  }
15  sprintf(filename, , myrank);
16  MPI_File_open(MPI_COMM_SELF, filename,
17  MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &myfile);
18  MPI_File_write(myfile, buf, BUFSIZE, MPI_INT,
19  MPI_STATUS_IGNORE);
20  MPI_File_close(&myfile);

```

```

21     MPI_Finalize();
22     return 0;
23 }
```

MPI функција која замењује функцију `fopen` назива се `MPI_File_open`:

```
MPI_File_open(MPI_COMM_SELF, filename,
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &myfile)
```

Аргументи ове функције су:

- **Комуникатор** - Ово је најзначајнија компонента У/И операција у MPI. Фајлови су отворени скупом процесора идентификованих од стране комуникатора. Ово обезбеђује да процеси раде на фајлу заједно, омогућавајући и комуникацију између процеса. Пошто сваки процес отвара свој фајл, користи се комуникатор `MPI_COMM_SELF`.
- **Назив фајла** - Други аргумент је стринг који представља назив фајла, као и у функцији `fopen`.
- **Тип мода** - Трећи аргумент је мод у коме је фајл отворен. У овом програму значи да је креиран или преписан ако већ постоји, као и да ће писање у фајл бити извршено само од стране овог програма. Константе `MPI_MODE_CREATE` и `MPI_MODE_WRONLY` представљају заставице.
- **`MPI_INFO_NULL`** - Предефинисана константа која представља лажну вредност за инфо аргумент `MPI_File_open`.
- **Фајл променљива** - Последњи аргумент је адреса `MPI_File` променљиве коју ће функција `MPI_File_open` отворити. Као и све MPI функције у C програму, `MPI_File_open` има повратну вредност. Уколико је фајл успешно отворен повратна вредност је `MPI_SUCCESS`.

Следећа функција која је важна за функционисање за MPI У/И је:

```
MPI_File_write(myfile, buf, BUFSIZE, MPI_INT,
               MPI_STATUS_IGNORE)
```

Податак који се уписује мора бити одређен адресом, величином и типом. Овим начином се описује бафер који ће бити коришћен за писање (Листинг II.4). То омогућава да се несуседни подаци у меморији запишу само једним позивом. Конкретно, овде се уписују `BUFSIZE` целих бројева са почетком на адреси `buf`. Последњи аргумент функције је статус, који је истог типа као и код `MPI_Recv`. У том случају занемариће се повратна вредност. MPI-2 дефинише специјалну вредност статуса `MPI_STATUS_IGNORE`. Ова вредност може бити послата као

аргумент било којој MPI функцији у циљу игнорисања повратне вредности. Технички, ово упрощење може побољшати ефикасност уколико нам статус заиста није потребан.

Функција `MPI_File_close` служи за затварање фајла. Послата адреса `myfile` биће преписана са `MPI_FILE_NULL` уколико се затварање фајла не обави успешно. Тако се могу идентификовати неважећи фајлови.

Листинг II.4: Програм са паралелним MPI функцијама

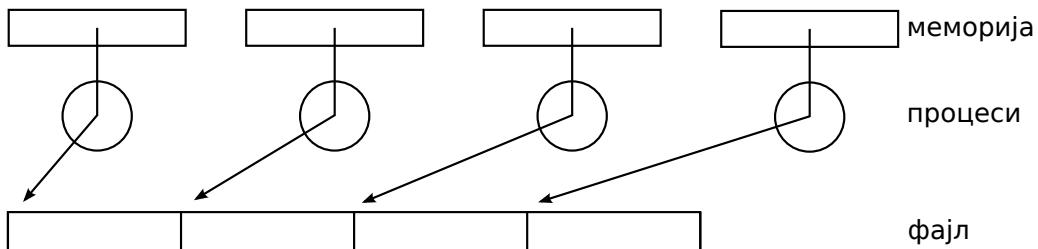
```

1 #include <mpi.h>
2 #include <stdio.h>
3 #define BUFSIZE 100
4 int main(int argc, char *argv[])
5 {
6     int i, myrank, buf[BUFSIZE];
7     MPI_File thefile;
8     MPI_Init(&argc, &argv);
9     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
10    for (i=0; i<BUFSIZE; i++)
11    {
12        buf[i] = myrank * BUFSIZE + i;
13    }
14    MPI_File_open(MPI_COMM_WORLD, ,
15    MPI_MODE_CREATE | MPI_MODE_WRONLY,
16    MPI_INFO_NULL, &thefile);
17    MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int),
18    MPI_INT, MPI_INT, , MPI_INFO_NULL);
19    MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,
20    MPI_STATUS_IGNORE);
21    MPI_File_close(&thefile);
22    MPI_Finalize();
23    return 0;
24 }
```

II.3.4 Паралелне MPI У/И операције са једним фајлом

Да би се добио још бољи учинак MPI У/И операција, потребно је изменити програм тако да процеси деле један фајл, уместо да пишу у више њих (Слика II.2). Тако се отклањају све мане код уписа у више фајлова и постиже се потпуни паралелизам.

Прва разлика у односу на програм који уписује у више различитих фајлова је први аргумент функције `MPI_File_open`. Пошто сада процес не приступа сопственом фајлу, већ једном дељеном, уместо комуникатора `MPI_COMM_SELF` користи се комуникатор `MPI_COMM_WORLD`. Тиме се постиже да сви процеси отварају исти фајл.



Слика II.2: Паралелне MPI У/И операције са једним фајлом

Ово је колективна операција на комуникатору, тако да сви процеси који учествују позивају `MPI_File_open`, при чему се, као што је напоменуто, отвара само један фајл. Део фајла се може видети у процесу, што се назива **погледом фајла**. Поглед фајла се поставља функцијом `MPI_File_set_view`:

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
                      MPI_Datatype etype, MPI_Datatype filetype,
                      const char *datarep, MPI_Info info)
```

Први аргумент идентификује фајл. Други аргумент је место (у бајтовима) у фајлу од кога почиње део фајла асоциран датом процесу. Овде множимо величину података `BUFSIZE * sizeof (int)` по рангу процеса, тако да поглед на сваки процес почиње на одговарајућем месту у фајлу. Тада аргумент је тип `MPI_Offset`, који на системима који подржавају велике фајлове очекује 64-битни цели број.

Следећи аргумент се назива **etype** погледа. То је скуп свих типова података који се налазе у фајлу. У овом случају то је `MPI_INT`, што значи да ће се у фајл увек уписати цели број. Наредни аргумент се назива `filetype`, и то је веома флексибилан начин да се опишу дисконтинуални погледи у фајлу. У овом случају ради се само о типу `MPI_INT`, тако да нема дисконтинуалних података за упис. Генерално, `etype` и `filetype` могу да буду било који предефинисани MPI типови података. Аргумент који означава представљање података у фајлу најчешће је типа стринг. Природно представљање значи да се подаци уписују у фајл тачно онако како су представљени у меморији. Ова шема чува податке и укупну ефикасност програма, јер се не губи време ни на какве додатне конверзије.

Остали типови представљања су *унутрашњи* и *external*¹³², који омогућавају различите врсте преноса између машина са различитим архитектурама. Последњи аргумент је инфо аргумент, као и у функцији `MPI_File_open`. Неке MPI функције у програмском језику С се налазе у Листингу II.5:

Листинг II.5: Основне MPI У/И функције

```

1 int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info
                   info, MPI_File *fh)
2
3 int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype
                      etype, MPI_Datatype filetype, char *datarep, MPI_Info info)
4
5 int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype
                     datatype, MPI_Status *status)
6 int MPI_File_close(MPI_File *fh)
7
8 int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
9 int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype
                    datatype, MPI_Status *status)

```

Овим начином, написани програм је независан од броја процеса на којима је покренут. Укупна величина фајла се добија тако да сваки процес ради са приближно истом величином података. MPI функција која се користи за добијање величине фајла је `MPI_File_get_size`. Први аргумент ове функције је отворени фајл, а други је адреса где треба сместити израчунату величину фајла у бајтовима. Пошто многи системи данас могу управљати фајловима чије су дужине превелике да би биле представљене као 32-битни цео број, MPI дефинише тип, `MPI_Offset` који може да садржи величину фајла у 64 бита. То је тип који се користи за све аргументе MPI функција које се односе на померање у фајловима. Програм за читање фајла је веома сличан оном који пише у фајл. Једина разлика између писања и читања је да процес не зна тачно колико ће података бити прочитано.

II.3.5 Коришћење појединачних фајл показивача

MPI програм са улазно/излазним операцијама је могуће писати и са појединачним фајл показивачима (Листинг II.6). Сваки од ових програма има део за У/И операције које отварају, читају и на крају затварају фајл. `MPI_File_open` је функција која отвара фајл. Први аргумент је комуникатор који идентификује групу процеса којима је потребан приступ фајлу. `MPI_COMM_WORLD` се користи зато што је свим процесима потребан приступ фајлу `/pfs/datafile`. MPI стандард не одређује формат за назив фајла. Свака од MPI имплементација има слободу да дефинише формат. Може се очекивати да ће имплементација подржати познате конвенције именовања. Имплементације које се покрећу на *Unix* окружењу подржавају *Unix* конвенцију именовања. У имплементацијама је назив директоријума опциони део назива фајла. Уколико не постоји, имплементација ће користити директоријум у коме се процес тренутно налази. Трећи аргумент функције `MPI_File_open` је начин приступа. У овом случају то је `MPI_MODE_RDONLY`, зато што је довољно да програм само чита из фајла. Четврти аргумент је инфо аргумент. Последњи аргумент је показивач на фајл који

враћа функција MPI_File_open.

Листинг II.6: MPI програм са појединачним фајл показивачима

```

1 #include <mpi.h>
2 #define FILESIZE (1024 * 1024)
3 int main (int argc, char **argv)
4 {
5     int *buf, rank, nprocs, nints, bufsize;
6     MPI_File fh;
7     MPI_Status status;
8
9     MPI_Init (&argc,&argv);
10    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
11    MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
12    bufsize = FILESIZE/nprocs;
13    buf = (int *) malloc (bufsize);
14    nints = bufsize/sizeof (int);
15    MPI_File_open (MPI_COMM_WORLD, , MPI_MODE_RDONLY, MPI_INFO_NULL, &
16                   fh);
16    MPI_File_seek (fh, rank*bufsize, MPI_SEEK_SET);
17    MPI_File_read (fh, buf, nints, MPI_INT, &status);
18    MPI_File_close (&fh);
19    free (buf);
20    MPI_Finalize ();
21
22    return 0;
23 }
```

С програм на Листингу II.6 извршава улазно/излазне операције користећи појединачне показиваче на фајл. После отварања фајла, сваки процес копира глобални фајл показивач у локални фајл показивач који показује на локацију у фајлу од које сваки процес чита свој део фајла. Први аргумент функције MPI_File_seek је показивач на фајл који је отворила функција MPI_File_open. Други аргумент одређује део фајла који сваки процес чита. MPI_SEEK_SET значи да се почетак локације за читање рачуна од почетка фајла. У програмском језику С за ово се користи већ поменути тип MPI_Offset. Део фајла сваког процеса се одређује производом ранга процеса и величине податка у бајтовима. Величина податка може се одредити и функцијама MPI_Get_count и MPI_Get_elements, користећи статус објекат који враћа функција MPI_File_read.

II.3.6 Употреба експлицитних одступања

MPI_File_read и MPI_File_write се базирају на појединачним фајл показивачима због тога што користе показивач на локацију у фајлу од које сваки процес чита фајл. MPI такође специфицира неколико функција које се називају **експлицитним функцијама одступања**. То су MPI_File_read_at и MPI_File_write_at. Ове функције не користе појединачне фајл показиваче.

У њима, локација у фајлу се директно прослеђује функцији као аргумент (Листинг II.7). Уколико више нити процеса приступају истом фајлу, онда се морају користити појединачни фајл показивачи због безбедности нити.

Листинг II.7: MPI функције за експлицитна одступања

```
1 int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int
                      count, MPI_Datatype datatype, MPI_Status *status)
2 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int
                      count, MPI_Datatype datatype, MPI_Status *status)
```

Уколико је потребно уписати податке у фајл, онда се користе функције `MPI_File_write` и `MPI_File_write_at`. Уместо заставице `MPI_MODE_RDONLY`, која је служила за читање фајла, за упис података у фајл се користе заставице `MPI_MODE_CREATE` и `MPI_MODE_WRONLY`. `MPI_MODE_CREATE` се користи за креирање фајла уколико он већ не постоји. `MPI_MODE_WRONLY` означава да је фајл отворен за писање. У програмском језику C, обе заставице могу се користити са битовним ИЛИ оператором: `MPI_MODE_CREATE | MPI_MODE_WRONLY`. Да би функција `MPI_File_open` креирала фајл, потребно је да постоји директоријум који је наведен у називу фајла.

II.3.7 Неконтинуални приступи и колективне У/И операције

У великом броју реалних паралелних апликација, сваком процесу је потребно да приступи малим деловима података који су смештени у фајлу неконтинуално, на пример [4, 17, 65, 77, 78, 85]. Један начин да се приступи неконтинуалним подацима је коришћењем функције за читање/писање малих континуалних делова, као у *Unix* системима. Због велике латенције улазно/излазних операција, приступање малим деловима података захтева пуно времена. Предност MPI над *Unix* системима је могућност приступа неконтинуалним деловима података pozивајући само једну функцију.

Неконтинуални приступи

MPI програм поседује концепт погледа на фајл. Поглед фајла у MPI-у је дефинисан као део фајла коме процес има приступ (Слика II.3). Користећи поглед на фајл, функције читања и писања могу приступити само том делу фајла. Сви остали подаци се прескачу. Када се отвори, фајл је доступан процесу и MPI трећира фајл као скуп бајтова (не као целе бројеве, реалне бројеве, итд.). Приликом покретања програма, сваки појединачни фајл показивач је постављен на 0. Ово се може променити помоћу функције `MPI_File_set_view` (Листинг II.8). Најчешће се то ради из два разлога:

- Да се одреди тип податка коме је потребно приступити, нпр. целим или реалним бројевима. Ово је нарочито битно за преносивост фајла, уколико

корисник жели да приступи фајлу са друге машине и са различитим представљањем фајла.

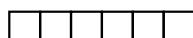
- Да се одреде делови фајла који ће бити прескочени, што је суштина неконтинуалног приступа. За појединачне фајл показиваче или експлицитна одступања, сваки процес може користити различит тип погледа.

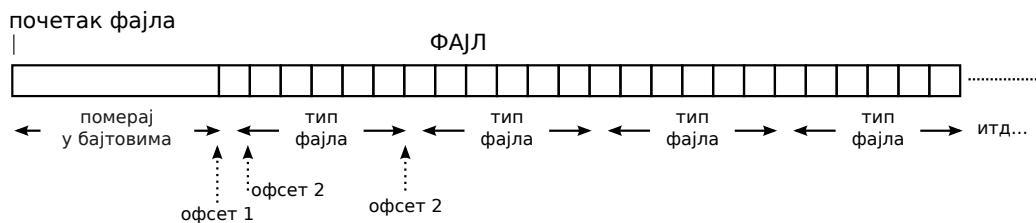
За приступ подацима са дељеним фајл показивачем потребно је да сви процеси користе исти поглед. Поглед фајла се може мењати небројено много пута. За одређивање погледа фајла користе се MPI типови података. Поглед је одређен следећим параметрима:

- Померај
- etype
- filetype

Померај одређује број бајтова који ће бити прескочени на почетку фајла. Користи се када је потребно да се прескочи читање заглавља фајла. Etype је основна јединица за приступ подацима. Може бити основни или изведени MPI тип података. Сви приступи фајлу се врше преко јединице типа etype. Сва одступања фајла се дефинишу као број etype-ова. Уколико је etype MPI_INT, појединачни и дељени фајл показивач биће померен за одређени број целих бројева. Filetype је основни или изведени тип података који дефинише који део фајла је доступан процесу и ког типа су подаци. Filetype мора бити исти као etype или изведен од типа који се заснива на etype. Поглед фајла почиње од помераја и састоји се од више суседних etype. Приликом отварања фајла, померај има вредност 0 и etype и filetype су типа MPI_BYTE. Ово је познато и као подразумевани поглед фајла.

etype = MPI_INT

 filetype = a contiguous type of 2 MPI_INT, resized to have 6 MPI_INT



Слика П.3: Поглед фајла

На Слици II.3 је приказан суседни изведен тип података који је представљен као два цела броја. Уколико се поставе још четири цела броја на крај овог типа података, функцијом MPI_Type_create_resized се добија тип податка који је величине шест целих бројева. Etype је типа MPI_INT, а померај је $5 * \text{sizeof}(\text{int})$. У MPI-1 верзији ово се изводи функцијом MPI_Type_struct.

Листинг II.8: MPI функције за неконтинуални приступ

```
1 int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype
   etype, MPI_Datatype filetype, char *datarep, MPI_Info info)
2 int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,
   MPI_Aint extent, MPI_Datatype *newtype)
```

Аргументи који се прослеђују функцији MPI_File_set_view су:

- Показивач на фајл
- Померај
- etype
- filetype
- Представљање података
- Инфо аргумент

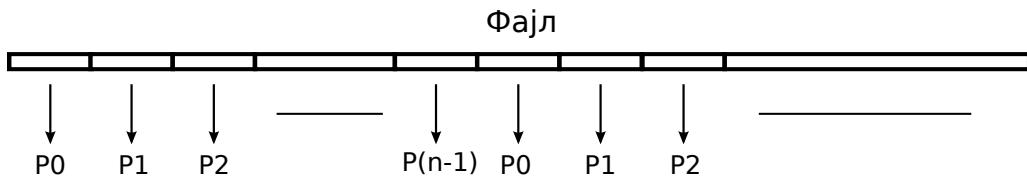
Подразумевано представљање је природно, док је подразумевани инфо аргумент MPI_INFO_NULL.

Колективне У/И операције

Разлика између колективних У/И операција са неконтинуалним приступом од других У/И операција је у томе што код колективних операција сваки процес чита мале блокове података, који се налазе у фајлу по принципу кружног распоређивања (Слика II.4). Са Unix У/И операцијама, једини начин да се читају подаци је читање сваког блока одвојено, због тога што Unix функције омогућавају приступ само једном континуалном делу података.

Код MPI-а, уместо позивања функције за читање више пута, може се дефинисати поглед на неконтинуални фајл сваког процеса, како би се прочитao фајл позивајући функцију само једном (Листинг II.9). Други начин је употреба колективног читања. MPI имплементација даје значајно бољи учинак у односу на Unix У/И функције.

FILESIZE одређује величину фајла у бајтовима. INTS_PER_BLK је величина сваког блока који процес треба да прочита (број целих бројева у блоку). Сваки процес треба да прочита неколико блокова у цикличном распореду.



Слика II.4: Колективне У/И операције

Листинг II.9: MPI програм са погледом фајла

```

1 #include <mpi.h>
2 #define FILESIZE
3 #define INTS_PER_BLK 104857616
4
5 int main(int argc, char **argv)
6 {
7     int *buf, rank, nprocs, nints, bufsize;
8     MPI_File fh;
9     MPI_Datatype filetype;
10
11    MPI_Init(&argc,&argv);
12    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
14
15    bufsize = FILESIZE/nprocs;
16    buf = (int *) malloc(bufsize);
17    nints = bufsize/sizeof(int);
18    MPI_File_open(MPI_COMM_WORLD, , MPI_MODE_RDONLY, MPI_INFO_NULL, &
19                  fh);
20    MPI_Type_vector(nints/INTS_PER_BLK, INTS_PER_BLK, INTS_PER_BLK*
21                    nprocs, MPI_INT, &filetype);
22    MPI_Type_commit(&filetype);
23    MPI_File_set_view(fh, INTS_PER_BLK*sizeof(int)*rank, MPI_INT,
24                      filetype, , MPI_INFO_NULL);
25    MPI_File_read_all(fh, buf, nints, MPI_INT, MPI_STATUS_IGNORE);
26
27    MPI_File_close(&fh);
28    MPI_Type_free(&filetype);
29    free(buf);
30    MPI_Finalize();
31
32    return 0;
33 }
```

Помоћу MPI_File_open отвара се фајл и поставља се комуникатор MPI_COMM_WORLD, пошто сваки процес треба да има приступ фајлу/pfs/datafile. Следећи корак је дефинисање погледа. За filetype, креира се изведени тип

типа вектор, користећи функцију `MPI_Type_vector`. Први аргумент ове функције је број блокова који сваки процес треба да прочита. Други аргумент је број целих бројева у сваком блоку, док је трећи број целих бројева између полазних елемената два узастопна блока које процес чита. Четврти аргумент је тип податка, у овом случају `MPI_INT`. Пети аргумент је повратна вредност функције `MPI_Type_vector`. Након креирања овог типа, нови тип се може користити као `filetype` аргумент функције `MPI_File_set_view`.

Etype је `MPI_INT`. Улазно/излазне операције се извршавају користећи колективну верзију функције `MPI_File_read`, која се назива `MPI_File_read_all`. У позивима ових функција нема суштинске разлике. Једина разлика је што се колективна функција позива од стране сваког процеса у комуникатору који је прослеђен функцији `MPI_File_open`. Овај комуникатор је имплицитно прослеђен функцији `MPI_File_read_all`. Функција `MPI_File_read` може се позивати независно од стране процеса.

II.3.8 Приступни низови смештени у фајловима

Пуно паралелних програма оперише са једним или више вишедимензионих низова дељених између процеса. Локални низ сваког процеса није континуално смештен у фајл. Свака врста низа сваког процеса је одвојена врстама локалних низова других процеса. MPI омогућава погодан начин за опис улазно/излазних операција које извршава преко једног јединог позива функције. Уколико корисник користи колективне У/И функције, MPI имплементација омогућава бољи учинак користећи овакав приступ, иако је сам приступ дисконтинуални. У MPI-2 је дефинисано два нова типа конструктора података: `darray` и `subarray`. Ове функције олакшавају креирање изведенih типова података, описујући локацију локалних низова спојених у један глобални низ. Ови типови података могу бити коришћени као `filetype` да опишу дисконтинуални приступ фајлу, када се обавља У/И операција за дељене низове.

II.3.9 Дељени низови

Конструктор типа података `darray` омогућава лак начин креирања изведеног типа података, који описује вишедимензиони глобални низ који се састоји од локалних низова (Слика II.5). Низ може бити било којих димензија и свака димензија може бити дистрибуирана на било који начин. Аргументи `darray` конструктора су величина низа, опис расподеле и ранг процеса чији је локални низ описан. Излаз је изведенти тип података који описује распоред локалних низова у глобалном низу. Постоје и други начини за креирање изведенih типова података, али су они нешто сложенији. Први аргумент функције `MPI_Type_create_darray` је број процеса којима је низ дистрибуиран. Други аргумент је ранг процеса чији је локални низ описан. Трећи аргумент су димензије низа,

док је четврти аргумент сам низ.

Листинг II.10: Део MPI програма са дељеним низовима

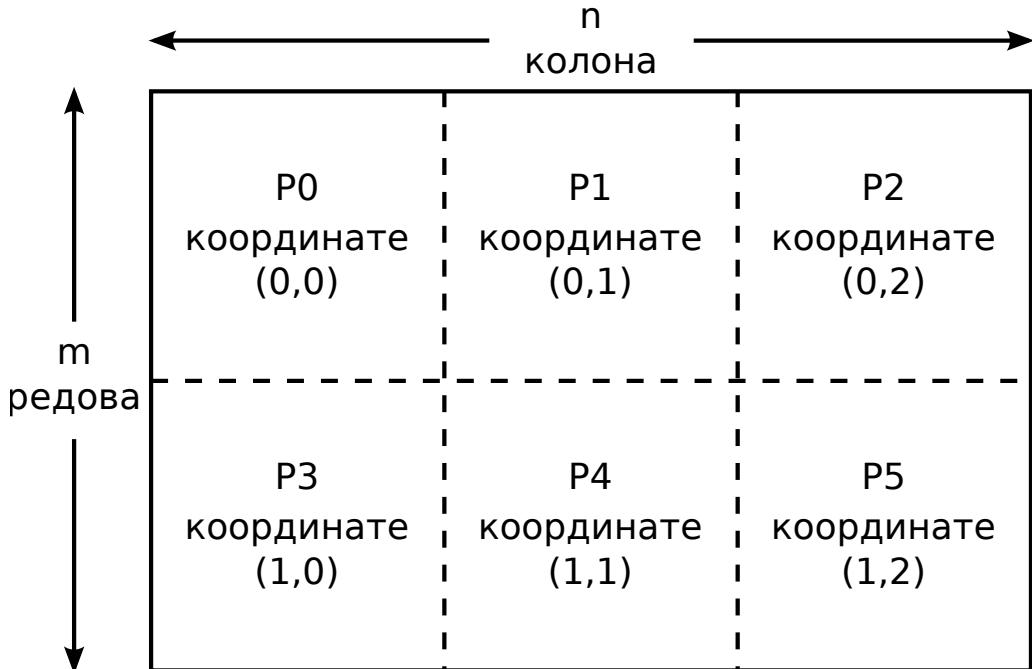
```

1 gsizes[0] = m;
2 gsizes[1] = n;
3
4 distribs[0] = MPI_DISTRIBUTE_BLOCK;
5 distribs[1] = MPI_DISTRIBUTE_BLOCK;
6
7 dargs[0] = MPI_DISTRIBUTE_DFLT_DARG; /* default block size */
8 dargs[1] = MPI_DISTRIBUTE_DFLT_DARG; /* default block size */
9 psizes[0] = 2;
10 psizes[1] = 3;
11
12 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13 MPI_Type_create_darray(6, rank, 2, gsizes, distribs, dargs, psizes,
   MPI_ORDER_C, MPI_FLOAT, &filetype);
14 MPI_Type_commit(&filetype);
15 MPI_File_open(MPI_COMM_WORLD, ,
16 MPI_MODE_CREATE | MPI_MODE_WRONLY,
17 MPI_INFO_NULL, &fh);
18 MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, ,
19 MPI_INFO_NULL);
20 local_array_size = num_local_rows * num_local_cols;
21 MPI_File_write_all(fh, local_array, local_array_size,
22 MPI_FLOAT, &status);
23 MPI_File_close(&fh);

```

Пети аргумент је низ који одређује начин на који је глобални низ подељен. На овом примеру, ту улогу има MPI_DISTRIBUTE_BLOCK. Шести аргумент одређује дистрибуциони параметар за сваку димензију, у овом случају k у цикличној (k) расподели. За блок и цикличне расподеле којима није потребан овај аргумент, подразумевана вредност је MPI_DISTRIBUTE_DFLT_DARG. Седми аргумент је низ који одређује број процеса дуж сваке димензије глобалног низа. Грид процеса увек има димензије као и глобални низ.

Осми аргумент функције MPI_Type_create_darray одређује редослед складиштења локалног низа у меморији, као и глобалног низа у фајлу. Девети аргумент је datatype, који описује ког је типа елемент низа, у овом програму MPI_FLOAT. Повратна вредност функције је изведенти тип података &filetype. После комитовања типа података, нови тип може се користити у постављању погледа (Листинг II.10).



Слика II.5: Дељени низ

II.3.10 Неблокирајуће У/И операције и дељене колективне У/И операције

MPI подржава неблокирајућу верзију независних функција за писање и читање. MPI механизам имплементира ове функције, слично као и неблокирајуће интерпроцесне комуникације. Називи неблокирајућих функције почињу префиксом `MPI_File_ixxx`, напр. `MPI_File_iread` и `MPI_File_iwrite_at`. Неблокирајуће функције враћају `MPI_Request` објекат. Могу се користити уобичајене `MPI_Test` и `MPI_Wait` операције. Коришћењем ових функција, може доћи до преклапања улазно/излазних операција са осталим комуникацијама у програму. Количина преклапања зависи од квалитета имплементације. За колективне операције, MPI подржава ограничен облик неблокирајућих операција, које се називају дељеним колективним У/И операцијама. Да би се користиле дељене колективне функције, корисник мора позвати функцију `MPI_File_read_all_end` како би се завршиле. Ограничено је да корисник у исто време може имати само једну дељену колективну операцију над једним фајлом. Дељене колективне функције не враћају `MPI_Request` објекат.

II.3.11 Дељени фајл показивачи

Постоји три начина да се одреди локација у фајлу са које је потребно прочитати или на коју треба уписати податке: појединачни фајл показивачи, експлицитна одступања и дељени фајл показивачи. **Дељени фајл показивач** је фајл показивач чија вредност је дељена између процеса који се налазе у комуникатору функције MPI_File_open. MPI спецификација обезбеђује функције MPI_File_read_shared и MPI_File_write_shared које читају/уписују податке са почетком у тренутној локацији дељеног фајл показивача. После позивања ових функција, дељени фајл показивач биће освежен новом количином података који су уписаны или прочитани. Следећи позив ових функција ће радити са освеженим дељеним фајл показивачем. Процес може експлицитно померити показивач дат у etypes јединицама помоћу функције MPI_File_seek_shared. MPI захтева да сви процеси имају исти фајл поглед када користе дељене фајл показиваче. За остала два начина могу се користити различити фајл погледи.

II.4 Даљински приступ меморији

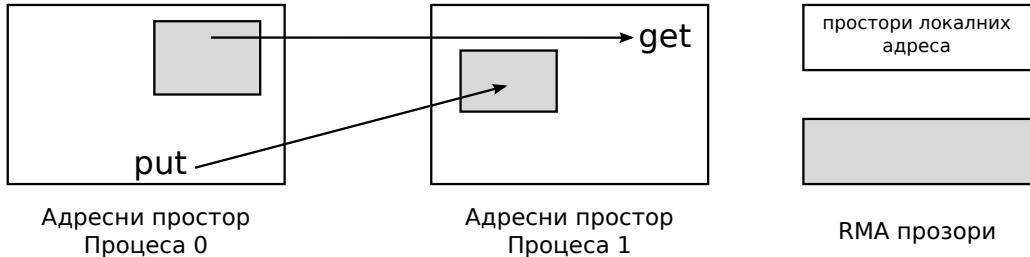
MPI-2 спецификација омогућава да процес директно приступи подацима другог процеса. Ове операције, које омогућавају читање и писање тих података називају се *Remote Memory Access* (RMA) операције. Главна карактеристика MPI имплементације је размена података између процеса помоћу операција за слање и примање података. Треба приметити да MPI-2 не омогућава реални дељени меморијски модел, али и да поред тога даљинске меморијске операције омогућавају велику флексибилност дељене меморије.

Даљински приступ меморији је пројектован да ради на машинама са дељеном меморијом и на окружењима која немају дељену меморију, као што су мреже радних станица које користе TCP/IP протокол за комуникацију. Њихова главна предност је флексибилност коју нуде у пројектовању алгоритама. Крајњи програми су преносиви кроз све MPI имплементације и биће ефикасни на свим платформама које омогућавају приступ меморији других процеса.

II.4.1 Меморијски оквири

У механизму строгог прослеђивања порука, бафери за слање и примање су одређени MPI типовима података који представљају делове адреса процеса који се шаљу другим процесима у случају слања, или адреса где ће други процеси уписати податке, у случају пријема. У MPI-2 спецификацији, појам комуникационске меморије је генерализован на појам оквира за даљински приступ меморији. Сваки процес може одредити део адресног простора који је доступан другим процесима за писање и читање. Операције писања и читања, које су покренуте од стране другог процеса називају се *get* и *put* операције за даљински приступ ме-

морији (Слика II.6). Трећи тип операција је `accumulate`. У MPI-2, оквир представља део меморије једног процеса који чини дистрибуирани објекат, који се назива оквирни објекат. Оквирни објекат је направљен од више оквира, од којих се сваки састоји од локалне меморијске области која је изложена другим процесима.



Слика II.6: Меморијски оквир

II.4.2 Перформансе програма са даљинским приступом меморији

Програм за рачунање броја π рачуна вредност помоћу нумеричке интеграције. У класичној верзији постоје два типа комуникације. Процес 0 комуницира са корисником и захтева број интервала за интеграцију. Помоћу функције `MPI_Bcast` процес 0 шаље тај број другим процесима. Сваки процес затим израчунава парцијалну суму и све суме се сумирају помоћу колективне операције `MPI_Reduce`.

У тзв. **једнострanoј верзији овог програма**, процес 0 снима вредност броја интервала као део RMA оквирног објекта, одакле га други процеси могу једноставно прочитати (Листинг II.11). После израчунања парцијалне суме, сви процеси додају своју вредност у други оквирни објекат помоћу операције `accumulate`. Сваки оквирни објекат садржи само један број у меморији. Оквирни објекти су представљени као променљиве типа `MPI_Win`. Функције за креирање оквира су следеће:

```

MPI_Win_create(&n, sizeof(int), 1, MPI_INFO_NULL,
               MPI_COMM_WORLD, &nwin)
MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL,
               MPI_COMM_WORLD, &nwin)

```

Позив са процеса 0 треба бити упарен са осталим процесима, иако они не доносе никакву меморију за оквирни објекат, пошто је `MPI_Win_create` колективна операција над процесима који се налазе у комуникатору. Комуникатор одређује који процеси могу приступити оквирном објекту. Прва два аргумента

функције MPI_Win_create су адреса и дужина оквира у бајтовима, Листинг II.11.

Листинг II.11: MPI програм са даљинским приступом меморији

```
1 #include <mpi.h>
2 #include <math.h>
3
4 int main(int argc, char *argv[])
5 {
6     int n, myid, numprocs, i;
7     double PI25DT = 3.141592653589793238462643;
8     double mypi, pi, h, sum, x;
9     MPI_Win nwin, piwin;
10    MPI_Init(&argc,&argv);
11    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
12    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
13    if (myid == 0)
14    {
15        MPI_Win_create(&n, sizeof(int), 1, MPI_INFO_NULL, MPI_COMM_WORLD
16                      , &nwin);
17        MPI_Win_create(&pi, sizeof(double), 1, MPI_INFO_NULL,
18                      MPI_COMM_WORLD, &piwin);
19    }
20    else
21    {
22        MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD,
23                      &nwin);
24        MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD,
25                      &piwin);
26    }
27    MPI_Win_fence(0, nwin);
28    while (1)
29    {
30        if (myid == 0)
31        {
32            printf();
33            fflush(stdout);
34            scanf(&n);
35            pi = 0.0;
36        }
37        MPI_Win_fence(0, nwin);
38        if (myid != 0)
39        MPI_Get(&n, 1, MPI_INT, 0, 0, 1, MPI_INT, nwin);
40        MPI_Win_fence(0, nwin);
41        if (n == 0)
42        {
43            break;
44        }
45        else
46        {
47            h = 1.0 / (double) n;
```

```

44     sum = 0.0;
45     for (i = myid + 1; i <= n; i += numprocs)
46     {
47         x = h * ((double)i - 0.5);
48         sum += (4.0 / (1.0 + x*x));
49     }
50     mypi = h * sum;
51     MPI_Win_fence(0, piwin);
52     MPI_Accumulate(&mypi, 1, MPI_DOUBLE, 0, 0, 1, MPI_DOUBLE,
53                     MPI_SUM, piwin);
54     MPI_Win_fence(0, piwin);
55     if (myid == 0)
56         printf("%f, %f, %f\n", pi, fabs(pi - PI25DT));
57     }
58 }
59 MPI_Win_free(&nwin);
60 MPI_Win_free(&piwin);
61 MPI_Finalize();
62 return 0;
63 }
```

Следећи аргумент је `displacement unit`, који се користи да одреди одступање локације у меморији. У овом примеру, сваки оквирни објекат садржи једну променљиву код које је одступање нула, тако да одступање може да се занемари. Четврти аргумент је `MPI_Info`, који се може користити да побољша учинак RMA операција. Пети аргумент је комуникатор који одређује скуп процеса који ће имати приступ меморији оквирног објекта. MPI имплементација враћа `MPI_Win` објекат као последњи аргумент. После позива функције `MPI_Win_create`, сваки процес који се налази у комуникатору има приступ подацима `nwin` помоћу операција `put`, `get` и `accumulate`. За меморију оквира није потребно алоцирати посебну меморију, већ се користи меморија самог процеса. MPI имплементација такође омогућава алоирање посебне меморије позивом функције `MPI_Alloc_mem`.

Други позив функције `MPI_Win_create` креира оквирни објекат `piwin`, дозвољавајући сваком процесу да приступи променљивој π првог процеса, где ће бити смештена израчуната вредност броја π . У следећем делу програма, процес са рангом 0 захтева број интервала, а затим остали процеси рачунају број π . Петља се завршава када корисник унесе нулу. Процеси којима ранг није нула, вредност броја n узимају директно из оквирног објекта без икакве додатне акције. Пре позива функције `MPI_Get` или било које функције за даљински приступ меморији, потребно је позвати функцију `MPI_Win_fence` да одвоји операције.

MPI имплементација омогућава специјални механизам синхронизације за операције дељене меморије. `Fence` операција је назvana функцијом `MPI_Win_fence`, која захтева два аргумента. Први аргумент је потврдни аргумент за дозвољавање оптимизације. Увек исправан потврдни аргумент има вредност 0. Други

аргумент је оквир на коме се операција извршава. MPI_Win_fence се може тумачити као баријера која одваја локалне операције на оквиру од скупа даљинских операција на оквиру. У овом програму одваја читање вредности променљиве n од осталих даљинских операција које следе. Вредност променљиве n остали процеси добијају позивом:

```
MPI_Get (&n, 1, MPI_INT, 0, 0, 1, MPI_INT, nwin)
```

Аргументи ове функције су слични аргументима функција које примају или шаљу податак. Get операција је слична операцији примања, па су зато прва три аргумента опис податка који се прима (адреса, количина и тип податка). Следећи аргумент је ранг процеса чијој меморији приступамо. У овом случају је ранг 0, јер сви процеси приступају меморији првог процеса. Следећа три аргумента дефинишу бафер за слање (адреса, количина и тип податка). Овде се адреса даје као одступање од почетка локације у дељеној меморији. У овом случају је 0, зато што се приступа само једној вредности. Последњи аргумент је објекат оквира. MPI_Get је неблокирајућа операција. После позива ове функције не може се гарантовати да је вредност смештена у променљиву n . Зато је потребно позвати MPI_Win_fence.

Сваки процес рачуна свој део суме $typi$. Сада се позива MPI_Win_fence, али на оквирном објекту piwin, како би се покренуо други RMA приступ. Позивом функције MPI_Accumulate, сабирају се све суме процеса у глобалну суму.

```
MPI_Accumulate(&typi, 1, MPI_DOUBLE, 0, 0, 1, MPI_DOUBLE,
MPI_SUM, piwin)
```

Прва три аргумента одређују локалну променљиву (адреса, количина и тип податка), док је четврти аргумент ранг процеса. Следећа три аргумента описују променљиву коју је потребно изменити. Аргумент који следи је операција коју је потребно извршити. Пошто нам је потребна глобална сумма, у овом случају то је MPI_SUM. Последњи аргумент је објекат оквира. Програм се завршава штампањем вредности броја π и ослобађањем меморије објекта помоћу функције MPI_Win_free. MPI_Win_free је колективна функција над комуникатором прослеђеног објекта оквира. Потписи коришћених MPI функција дати су у Листингу II.12.

Листинг II.12: MPI функције

```
1 int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
MPI_Info info, MPI_Comm comm, MPI_Win *win)
2 int MPI_Win_fence(int assert, MPI_Win win)
3 int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
origin_datatype,
4 int target_rank, MPI_Aint target_disp, int target_count,
MPI_Datatype target_datatype, MPI_Win win)
```

```

5 int MPI_Accumulate(void *origin_addr, int origin_count, MPI_Datatype
   origin_datatype, int target_rank, MPI_Aint target_disp, int
   target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win
   win)
6 int MPI_Win_free(MPI_Win *win)

```

II.5 Управљање процесима

Процес модел који се користи у MPI-1 имплементацијама користи фиксиран број процеса током MPI рачунања. Ово је концептуално једноставан модел, јер ставља све сложености интеракције са оперативним системом (који мора бити укључен у стварање процеса) потпуно изван оквира апликације. Када се изврши MPI_Init, процеси су покренути и комуникатор MPI_COMM_WORLD поставља задати број процеса. Они могу међусобно да комуницирају преко комуникатора. Други комуникатори имају своје групе, које чине подгрупе MPI_COMM_WORLD. Динамичнији приступ управљањем процесима пролазилази из PVM (*Parallel Virtual Machine*) заједнице [10], где се процеси покрећу под контролом апликације. Интеркомуникатор служи да повеже две групе процеса. Интеркомуникатори омогућавају природан начин да опишу *spawning* процесе. Интеркомуникатори се могу спојити помоћу функције MPI_Intercomm_merge, чије је повратна вредност нови интеркомуникатор.

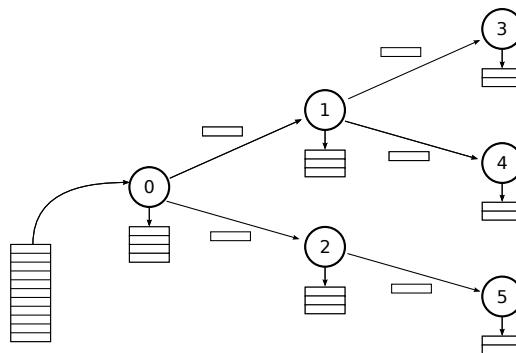
II.5.1 Креирање процеса

У MPI-2 имплементацијама, процес се креира помоћу функције MPI_Comm_spawn. Кључна предности MPI_Comm_spawn су:

- Ово је колективна операција над новим процесима.
- Нови процеси имају свој властити MPI_COMM_WORLD.
- Функција MPI_Comm_parent, позвана од стране процеса потомка, као повратну вредност има интеркомуникатор који садржи потомке процеса као локалну групу и родитеље као даљинску групу.

II.5.2 Пример паралелног копирања

На Листингу II.13 дат је једноставни услужни програм који извршава паралелно копирање тако што копира фајл са локалног диска машине на локалне дискове других машина. Са MPI имплементацијом може се на скалабилан начин умањити време извршавања програма. Основни начин слања фајла помоћу

Слика II.7: Пример паралелног копирања (*broadcast*)

MPI-а је помоћу функције `MPI_Bcast` за слање са `root` процеса (Слика II.7). Да би се покренуо програм, потребно је на свакој машини имати извршни фајл.

Процес са рангом 0 чита фајл, а затим користећи `MPI_Bcast` шаље блок по блок фајла другим процесима. Овај начин садржи три облика паралелизма:

1. Сви процеси извршавају паралелне улазно/излазне операције са фајлом.
2. Већи део слања порука између процеса се одвија паралелно.
3. Подела фајла у блокове се одвија у паралелизму типа цевовода.

Први део програма је парсирање листе свих машина на које је потребно ис-копирати фајл, а затим и креирање празног фајла на тим машинама. Функција `makehostlist` парсира први аргумент и штампа списак машина у фајл чији је назив прослеђен као други аргумент. Број машина је повратна вредност ове функције.

```
makehostlist( argv[1], "targets", &num_hosts );
```

Да би сви процеси знали име фајла у коме се налази списак машина, потребно је да се назив фајла проследи функцији која покреће нове процесе `MPI_Comm_spawn` помоћу инфо објекта. Програм је приказан на Листингу II.13.

Листинг II.13: MPI програм за паралелно копирање

```

1 #include <mpi.h>
2 #include <stdio.h>
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <fcntl.h>
6 #define BUFSIZE 256*1024
7 #define CMDSIZE 80
8
9 int main( int argc, char *argv[] )
```

```

10 {
11     int num_hosts, mystatus, allstatus, done, numread;
12     int infd, outfd;
13     char utfilename[MAXPATHLEN], controlmsg[CMDSIZE];
14     char buf[BUFSIZE];
15     char soft_limit[20];
16     MPI_Info hostinfo;
17     MPI_Comm pcpslaves, all_processes;
18
19     MPI_Init( &argc, &argv );
20     makehostlist( argv[1], "targets", &num_hosts );
21     MPI_Info_create( &hostinfo );
22     MPI_Info_set( hostinfo, "file", "targets" );
23     sprintf( soft_limit, "0:%d", num_hosts );
24     MPI_Info_set( hostinfo, "soft", soft_limit );
25     MPI_Comm_spawn( "pcp_slave", MPI_ARGV_NULL, num_hosts, hostinfo,
26                     0, MPI_COMM_SELF, &pcpslaves, MPI_ERRCODES_IGNORE );
27     MPI_Info_free( &hostinfo );
28     MPI_Intercomm_merge( pcpslaves, 0, &all_processes );
29     strcpy( outfilename, argv[3] );
30     if ( (infd = open( argv[2], O_RDONLY ) ) == -1 )
31     {
32         fprintf( stderr, "input %s does not exist\n", argv[2] );
33         sprintf( controlmsg, "exit" );
34         MPI_Bcast( controlmsg, CMDSIZE, MPI_CHAR, 0, all_processes );
35         MPI_Finalize();
36         return -1 ;
37     }
38     else
39     {
40         sprintf( controlmsg, "ready" );
41         MPI_Bcast( controlmsg, CMDSIZE, MPI_CHAR, 0, all_processes );
42         MPI_Bcast( outfilename, MAXPATHLEN, MPI_CHAR, 0, all_processes );
43         if ( (outfd = open( outfilename, O_CREAT|O_WRONLY|O_TRUNC,
44                           S_IRWXU ) ) == -1 )
45         {
46             mystatus = -1;
47         }
48     else
49     {
50         mystatus = 0;
51     }
52     MPI_Allreduce( &mystatus, &allstatus, 1, MPI_INT, MPI_MIN,
53                   all_processes );
54     if ( allstatus == -1 )
55     {
56         fprintf( stderr, "Output file %s could not be opened\n",
57                 outfilename );
58     MPI_Finalize();
59     return 1 ;

```

```

58     }
59     /* at this point all files have been successfully opened */
60     done = 0;
61     while (!done)
62     {
63         numread = read( infd, buf, BUFSIZE );
64         MPI_Bcast( &numread, 1, MPI_INT, 0, all_processes );
65         if ( numread > 0 )
66         {
67             MPI_Bcast( buf, numread, MPI_BYTE, 0, all_processes );
68             write( outfd, buf, numread );
69         }
70         else
71         {
72             close( outfd );
73             done = 1;
74         }
75     }
76     MPI_Comm_free( &pcpslaves );
77     MPI_Comm_free( &all_processes );
78     MPI_Finalize();
79
80     return 0;
81 }
```

Програм конвертује комуникатор pcpslaves који садржи покренут процес и процесе које је креирала функција MPI_Comm_spawn у један заједнички интеркомуникатор помоћу MPI_Intercomm_merge. Интеркомуникатор all_processes се користи као комуникатор између root чвора и осталих чворова. Процеси покушавају да отворе улазни фајл, и уколико дође до грешке, шаље се сигнал за прекид рада.

Да би се знало да је сваки процес отворио фајл, користи се функција MPI_Allreduce са операцијом MPI_MIN. Уколико било који процес не може да отвори фајл, сви процеси ће то сазнати и позвати MPI_Finalize за прекид рада. Код за процес потомак је сличан коду процеса родитеља, с тим што потомак мора да позове MPI_Comm_get_parent функцију да успостави контакт са родитељем. Потомак не обрађује аргументе нити штампа поруке. Процес родитељ затим чита блок по блок фајла и шаље осталим процесима. На крају сви процеси ослобађају интеркомуникатор креиран од стране MPI_Comm_spawn и обједињени интеркомуникатор. Главна разлика између функције MPI_Comm_spawn и осталих система за слање порука је природност колективних операција. У MPI имплементацији, група процеса колективно креира другу групу процеса који су међусобно синхронизовани. Тиме се спречавају додатни услови и омогућава неопходна комуникациона инфраструктура.

Литература

- [1] Michael Jay Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill Higher Education, 2004.
- [2] Andrew Tanenbaum, Computer Networks 4th, Prentice Hall Professional, 2002.
- [3] <https://www.quora.com/Why-should-we-care-about-exaflop-speed-supercomputers>, Jyh 2016.
- [4] Avi Kavas, David Er-El, and Dror G. Feitelson, „Using multicast to pre-load jobs on the ParPar cluster“. Parallel Computing 27(3) pp. 315-327, Feb 2001.
- [5] N. Petkov: Systolic Parallel Processing; North Holland Publishing Co, 1992.
- [6] G. W. Recktenwald, „Finite-Difference Approximations to the Heat Equation“, Portland, Oregon, USA, 2011.
- [7] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. The International Journal of Supercomputer Applications and High Performance Computing, 8, 1994.
- [8] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface standard. Technical Report, 1997.
- [9] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard Version 3.1 June 4, 2015.
- [10] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing MIT Press Cambridge, MA, USA 1994.
- [11] Schroeder, L. (1974). Buffon's needle problem: An exciting application of many mathematical concepts. Mathematics Teacher, 67 (2), 183-186.
- [12] P. D. Coddington. Random number generators for parallel computers. Technical report, Northeast Parallel Architectures Center, Syracuse University, Syracuse, New York, 1997.

- [13] Brueckner, Rich (August 6, 2014). „Why Chapel for Parallel Programming?“. Insi-deHPC. Retrieved 2015-03-23.
- [14] „Julia Documentation“. julialang.org. Retrieved 18 November 2014.
- [15] <http://spark.apache.org/docs/latest/>, Август 2016.
- [16] Dennis and Misunas, „A Preliminary Architecture for a Basic Data Flow Processor“, ISCA 1974.
- [17] Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. In Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 2004.

Издавач
Природно-математички факултет Крагујевац
Радоја Домановића 12
<http://www.pmf.kg.ac.rs>
Крагујевац

СИР – Каталогизација у публикацији – Народна библиотека Србије, Београд

004.42.032.24(075.8)

ИВАНОВИЋ, Милош, 1978–

Паралелно програмирање : скрипта са примерима : базирано на: Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP* / Милош Ивановић. – Крагујевац : Природно-математички факултет, 2016 (Крагујевац : Сквер). – 190 стр. : илустр. ; 25 cm

Тираж 300. – Библиографија: стр. 189-190.

ISBN 978-86-6009-042-5

а) Паралелно програмирање
COBISS.SR-ID 226920204
