**Wa-Tor**

Проект за курса „Системи за паралелна обработка“

**Изготвил**: Ива Караджова, КН, ф.н. 81773

**Дата:** 7 юли 2021

Table of Contents

[Постановка на задачата 2](#_Toc76548415)

[Анализ на съществуващи реализации 4](#_Toc76548416)

[Използван технологии 6](#_Toc76548417)

[Реализация на проекта 7](#_Toc76548418)

[Тестване на проекта 18](#_Toc76548419)

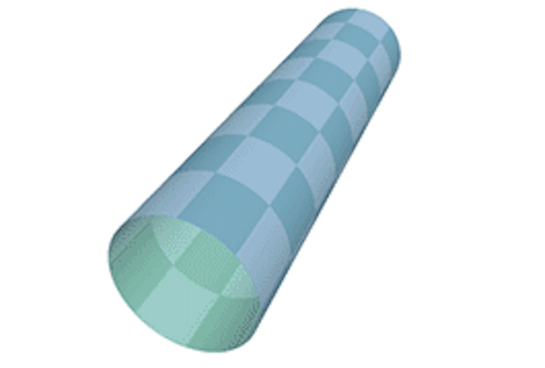
[Литература 24](#_Toc76548420)

# Постановка на задачата

Описание на Wa-tor

Wa-tor представя симулация на екологичната война между риби и акули, обитаващи общ воден басейн. Съставена е от Александър Деудни през 1984 и бива представена в американското научно списание „*Scientific American“.*

Моделът му представя движението на отделните представители на двата вида животни както и отразява промяната на популациите им спрямо основните жизнени процеси. Деудни описва света на водните обитатели като геометричната форма тор (форма на геврек, описана при завъртането на окръжност около ос, лежаща в нейната равнина). По този начин обитателите не могат да напуснат теоретичния си хабитат. В реализацията си той представя тора като двуизмерна матрица от клетки, в която всяка позиция може да бъде заета от риба, акула или да е празна (т.е. да има само вода). За да бъде напълно акуратно представянето на тор като двумерен обект, ако някое животно премине границите на матрицата, то автоматично се възстановява на противоположния край. По този начин е опростена безкрайността на фигурата тор.

Shape

Description automatically generated with medium confidenceBackground pattern

Description automatically generated with low confidence

В света на Wa-Tor събитията се извършват на дискретни интервали от време наречени *Chronons*, чиято конкретна дължина не влия на симулацията и поради тази причина няма нужда да бъде специфицирана. На всеки интервал могат да настъпят набор от събития, които влияят на популацията.

Правила на симулацията

В началото на симулацията се генерира двумерно правоъгълно поле от клетки, като по произволен начин се разпределят даден брой риби и акули, всичко останало се приема, че е запълнено от вода, т.е. е празно. Задават се също енергия за рибите, гладна смърт за акулите и възраст за размножаване на риби и на акули.

Рибите могат да се преместят в съседна клетка, която не е обитавана в същия момент от акула или друга риба, но само ако имат достатъчно останала енергия за това, в противен случай умират. Ако дадена риба е стигнала възрастта за размножаване, тя създава нова риба в свободна съседна клетка. Тази възраст се „рестартира“ и след същия период от време може отново да се размножи.

Акулите също се местят в съседна клетка на всеки интервал от време, като за тях приоритетни са клетките, обитавани от риби, ако такава не е налична, акулата може да се премести в свободна от други акули клетка. Ако акула успее да заеме клетка с риба, тя изяжда плячката и увеличава гладия си индекс. Този индекс, както енергията при рибите, показва още колко интервала от време акулата може да живее без да погълне риба. Сходно с рибите е и размножаването при акулите. Отново, когато акула достигне възрастта си на размножаване тя създава малко в съседна клетка, като след това тази възраст се „рестартира“ и акулата след същия период от време може отново да се възпроизведе.

Използваното понятие съседна клетка означава всяка клетка, която е има обща стена с текущо разглежданата. Съседната клетка се избира сред възможните на произволен принцип. Ако не съществуват клетки, към които да премине животното, то остава на място. [1]

Условие на проекта

Идеята зад проекта е да се разработи симулация на света на Wa-Tor, следвайки поставените правила. Итеративния алгоритъм на симулацията следва да бъде ускорен чрез паралелизация. За тази цял трябва да бъде подбран подходящ начин за разбиване на работа по проекта за всяка паралелна единица на програмата и също така да се състави начин за комуникация между отделните единици.

# Анализ на съществуващи реализации

В реализацията на *Overeinder, B. J., & Sloot, P. M. A.* Wa-tor директно се представя като *асинхронен клетъчен автомат* (“*asynchronous cellular automaton*”). Всяка една позиция от двуизмерното табло на wa-tor може да бъде представена като отделен клетъчен автомат, чиито състоянията спазват по-рано описаните правила на симулацията. Всяко състояние отговаря на различна ситуация спрямо състоянието на съседните клетки и променливите за животното, което се намира в сегашния интервал в съответната клетка.

За паралелизацията на алгоритъма е избрано да се разбие множеството от обработваеми единици – в случая това са клетките на таблото. Океанът се разделя на под домейни, по-малки части с приблизително равна площ, всяка от които се обработва от отделен процес.

Под домейните на океана се нуждаят комуникация в случаите, когато трябва риба или акула да премине граници на съответния домейн. В конкретната реализация този проблем е преодолян чрез изпращане на съобщения между процесите. Процес информира съседа си, когато риба премине в частта от таблото, която той обработва в чужда. По този начин може да избегнат конфликти в крайните части на всяка обработваема област, тъй като без комуникацията два процеса, действащи върху съседни област могат да извършат забранен ход, като пример да преместят риба върху клетка, където вече има друга.

Синхронизацията на процесите се осъществява чрез метода *Time Wrap*. При него всяка интеракция между процесите се отбелязва с времева отметка (*time stamp)*. Оставя се да продължава изпълнението на процесите докато не настъпи грешка във времевите отметки, т.е. настъпва комуникация между процеси с времева отметка по-малка от предходна. В тази ситуация се изпълнява *rollback* и последните действия до аномалията във времевите отметки не бъде премахната. [2]

Голяма част от откритите реализации използват същата идея за разделяне на океана на по-малки участъци. В реализацията на *Nicolo Tonci* отново се среща тази метод. Разликата тук се състои в справянето с е конфликтите на границите на отделните под части на океана, които се обработват от различни процеси. В случая вместо отбелязване на времето на всяка комуникация се използват свойствата на семафорите. Като общ ресурс се разглежда картата на океана, обработвана от отделните единици. Когато процес достигне критичната част от площта, за която отговаря, той бива приспан от мутекс, ако друг процес в същия момент работи по своя гранична част. Така като се предоставя единичен достъп до критичните части на океана се предотвратяват конфликти при обработката им, защото няма как два процеса да извършат противоречащи ходове на една клетка. [3]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Num. | Модел | Балансиране | Технология | Синхронизация |
| 1. | MPMD (решетка) | Статично | C++ | Time Wrap + messaging |
| 2. | MPMD (решетка) | Статично | C | Semaphores |

Вижда се, че двата проекта предприемат много сходен подход към ускоряването на симулацията. И при двата се наблюдава разбиване на цялото поле на океана на подмножества от клетки, които биват обработвани самостоятелно от отделни процеси. При този подход съществуват чисто логически ограничения при паралелизацията. Ако се абстрахираме от възможностите на тестовата машина, за целите на проекта няма как да бъдат създадени повече процес от броя на клетки в полето на океана. Дори и в тази ситуация толкова голям брой процеси не е удачен.

Поради сходността на решенията подходът към разработвания проект ще бъде много подобен. В реализациите на двата проекта-източници, не бива спомена оптимален брой използвани процеси. За да стигнем до този оптимален брой в текущия проект ще бъдат направени тестове за установяването им.

# Използвани технологии

Голяма част от реализациите на Wa-tor са направени на C/C++. Езикът предоставя една от най-бързите обработки на данни и работи на достатъчно ниско ниво, което предоставя на потребителя гъвкавост при разработката на програми. Също така поддържа паралелизъм както чрез създаването на процеси и така и под формата на корутини.

Негов по-осъвременен вариант е Golang. Този език също предоставя бърза компилация, способи за работа на ниско ниво и поддръжка на паралелизъм. Избран е за реализацията на проекта, поради няколко причини.

1. Има богат набор от библиотеки, свързани с обработката на изображения, необходимо за дебъгване и визуализацията на света на Wa-tor. Използвана е *“image/gif”.*
2. Ефективен метод за следене и запазване на *benchmark* тестове, чрез които може да се следи представянето на кода при паралелно изпълнение.
3. Предоставя *garbage collection*
4. Има способ за конкурентно изпълнение и синхронизация – goroutines.

За разлика от C++, Golang има много средства от по-високо ниво за синхронизация на рутини, което помага за избягването на програмистка грешка при разработка. Също така предоставя load balancing за рутините. Чрез него се подсигурява, че ако дадена нишка приспи процесите в опашката си например заради четене от файл, чакащите ще бъдат преместени върху работеща нишка. Проектът не се очаква да извършва конкурентно дейност, която да изисква приспиването на нишка, но предотвратява при локално тестване въздействието на външни процеси.

# Реализация на проекта

Следващите страници описват техническите детайли по реализацията на проекта. Включват анализ на структурите, в които се съхранява информация за света на Wa-tor и описание на действието на основните действащи функциите. Освен това целта на предстоящите страници е да представи идеята за ускорение на симулацията, заедно с някои проблеми, възникнали при разработката.

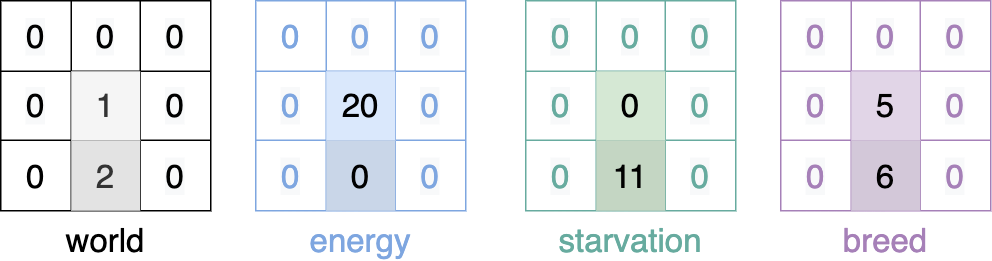
Структурно описание

Света на Wa-tor основните обекти, които трябва да бъдат представени чрез програмни структури са океанът, рибите и акулите. Добрите практики насочват към ООП подход към задачата. При такъв подход би се очаквало да се състави интерфейс за водните обитатели, който да съдържа общите им действия като размножаване и смърт. Този интерфейс след това може да бъде наследен от клас риба и клас акула, които да реализират собствените си методи за разплод и край на живота. За съжаление, в случая не се нуждаем от подобна скалируемост на задачата, тъй като още от началото знаем за цялата разновидност от обекти, с които ще се сблъскаме.

Затова в тази задача е приет по алгоритмичен подход. Представяме океанът като двумерна матрица, като във всяка една нейна клетка може да попадне риба или акула. Тази матрица е именувана – world. Тя е съставена от осембитови целочислени стойности, като всяка клетка може да съдържа конкретно 2, 1 или 0, съответно отговарящи на акула, риба или празна клетка. За движението на животните и интеракцията помежду им итерираме матрицата и за всяко животно, на което попаднем прилагаме правилата за движение на съответния вид, описани по-рано. Като резултат номера на водния обитател, 1 или 2, се премества в позволена съседна клетка, като предходната получава стойност 0, т.е. се запълва от вода.

Чрез този по-семпъл модел можем лесно и да визуализираме взаимодействието на риби и акули, като представим всяка клетка на матрицата като набор от 4 съседни пиксела, подредени в квадрат. Всяко от тези малки множества от пиксели се оцветява в един цвят зависимост от стойността на клетката, която ги определя. В реализацията при 0 се задава син цвят на пикселите, при 1 – тюркоазен, при 2 – лилав. Друго предимство на матриците е че имаме достъп до всеки елемент за константно време. Това е може би водещата мотивация за техния избор.

Всяко животно се асоциира и с набор от стойности: рибите с останала енергия и време до разплод, а акулите с брой интервали от време до гладна смърт и до размножаване. За съхранение на тези стойности стойности създаваме идентични на world матрици – *starvation, energy и breed* . На същите индекси на клетките в *world*, върху които има риба, в *energy* се пази оставащата й енергия, а в *breed* – интервалите до размножаване. Аналогично за акулите, на индексите в *world*, съответни на клетка с двойка, се запазва текущото ниво на глад на акулата, а в *breed* – интервалите до размножаване.



Изображение демонстрира връзката между четирите таблици. В матрицата world виждаме, че *wa-tor* света е популиран с една риба на позиция и една акула. Тъй като акулата не притежава енергия, нейната стойност в матрицата *energy* e 0, поради същата причина стойността в *starvation* на рибата също е 0. Двете матрици *starvation и energy* могат да бъдат обединени в една, но за четимост са оставени като отделни.

Функционално описание

Функционалностите на проекта могат да бъдат разделени на две основни части – инициализацията и симулация.

Инициализация

В частта на инициализация се създават матриците и се заселват с риби и акули. За целта програмата се нуждае от няколко константни стойности:

* ***length*** *– дължината на океана*
* ***width*** *– широчината на океана (= 2 \* length)*
* ***fishBreed*** *– възраст размножаване на рибите*
* ***sharkBreed*** *– възраст за размножаване на акулите*
* ***starvationTime*** *– индекс на гладна смърт (интервали от време преди акулата да умре, ако не погълне риба)*
* ***energyTime*** *– енергия на рибите (интервали от време до тяхната смърт)*
* ***numFish*** *– първоначален брой риби*
* ***numSharks*** *– първоначален брой акули*

Тези стойности са записани в програмата и могат да се регулират от програмния код. Като евентуално подобрение на проекта – тези параметри могат да се изнесат в конфигурационен файл, за да бъдат променяни безопасно от потребители.

Инициализация на всички структури нужни за представянето на океана и обитателите му се генерират от функцията:

**func initWorld()**

тя инициализира матриците и добавя животните заедно с техните специфики, като използва описаните константи за размера на матриците и броя и характеристиките на животните.

Първоначалното разположението на водните обитатели избира на произволен принцип. За целта извиква функцията –

***func addAnimal(num int, animal byte)***

задава чрез *num* броя на животните, които искаме да се генерират, и вида животно чрез *animal.* Тя генерира два масива с дължина num от произволни стойности в границите на океана, като първия задава на x кoордината на всяко животно, а втория – y коoрдината.

Симулация

При симулацията на света на *Wa-tor*  основната дейност се извършва от функцията:

***func nextChronon()***

тя симулира действията на водните обитатели за една единица време (в Wa-tor единицата време е прието да се нарича *Chronon*). Итерира през матрицата *world* и когато попадне на риба или акула, извиква хендлъра за съответното животно.

***func handleFish(x int, y int)***

Тя търси в коя посока да премести рибата като съставя списък от свободни съседни позиции на (x, y) в матрицата *world* и избира произволен елемент от този списък. Грижи се също да премести стойностите в таблиците energy и breed, като ги намали с единица. В случай, че energy стигне 0, рибата бива убита чрез ***killAnimal****, а ако breed* стигне 0 се създава нова риба на позиция (х, у), като се попълват полетата във всички матрици, а времето за репродукция breed на родителската риба се рестартира.

***func handleShark(x int, y int)***

има аналогично действие. Също отговаря за преместването на акулата на нова позиция и за обновяването на характеристиките й. Особеното при акулата е, че тя с приоритет отива към позиция с храна, т.е. в която има риба, затова функцията първо съставя списък от съседни позиции с риби и ако не е празен избира произволна от тях, към която да премести акулата. В този случай индексът за глад на акулата се увеличава с единица, а рибата бива убита. Когато няма риба в съседна клетка, отново се съставя списък, но този път от свободните позиции и се избира произволна от тях. Акулата в този случай „гладува“, затова индексът за гладна смърт (*starvation*) се намаля с едно. Breed стойността при всички ситуации се намаля с 1 и както при рибите, ако достигне нула, се създава нова акула на старата позиция.

Интересен е случаят по краищата на матрицата. Както беше споменато в началото, матрицата е репрезентация на геврекообразната фигура тор. Следователно, когато някой воден обитател е на крайна колона или ред, трябва да има възможност да продължи и в съответния противоположен край. Като за генериране на правилните координати от срещуположния края, на които трябва да се появи животното, се грижи функцията:

***func getDirections(x int, y int)***

*handleFish u handleShark*  я използват, за да се подсигури овалността на фигурата.

В следващата диаграма са изобразени основните действащи и връзките между тях. Помощните функции са комбинирани, тъй като представляват набор от много семпли функции, които спомагат с четимостта на кода.

Text

Description automatically generated

За да бъде завършена симулацията, трябва да бъдат изпълнение повече от една стъпка. Следователно последната част от симулацията е функцията *nextChronon* да бъде извикана многократно*.* Програмата предоставя и възможност всяка стъпка да бъде визуализирана.

A close up of a flower

Description automatically generated with low confidence

1Графично представяне на Wa-tor света

Паралелизация

Модел на паралелния алгоритъм

За паралелното изпълнение на симулацията се използва модела MPMD (Multiple Program Multiple Data) с топология решетка. Решетката е най-подходяща при двумерни матрици с равномерно разпределение на информацията. Тъй като в случая животните са разпределени произволно из картата на океана, в голяма част от случаите те ще бъдат равномерно разпределени. Следователно решетката е подходяща топология за този проект.

За да приложим MPMD модела, разбиваме матриците на подматрици, като всяка подматрица бива обработвана от отделна рутина. Главната рутина върху, която се стартира main функцията на кода, няма да обработва части от света на wa-tor, тя само ще разпределя задачите за останалите, ще стартира другите рутини и ще обработи отговора от всяка тях.

За да се позволи тестване на кода с различен брой рутини, е добавен параметър ***numRoutines*** към функцията изпълняваща цялата симулация. След като програмата има броя рутини, които трябва да създаде, следващата й задача е да раздели задачите между тези рутини. Както беше споменато по-горе, разпределението на водните животни клони към равномерно, затова се налага матриците world, energy, starvation и breed да бъдат разделени на равни участъци.

В реализацията матриците биват разделени на вертикални ивици всяка, от които е ширина *length/numRoutines,* с изключение на последната , която е с ширина остатъка при делене на двете стойности. За да може да се приспособява функцията *nextChronon()* към променящите се координати на матрицата, която обхожда, му се добавят параметри за размерите на матрицата : ***func nextChronon(length int, widthR int, widthL int).***

Main Routine

Routine4

Routine3 1

Routine2223 1

Routine6

Routine 5

Routine1223 1

Балансиране

В случая на Wa-tor балансирането на задачите изпълнявани от рутините е статично. Всяка рутина покрива животните намиращи се в подадената й област. При динамично балансиране можеше проекта да е допълнен с логика за равномерно разпределяне на животните, които трябва да се преместят на всеки интервал, таке че и товара по обработката да бъде равномерен и да предотврати дълги почивки на някои от рутините. Например нека разгледаме какво ще се случи при изпълнение разделено на две рутини. Те ще обработват съответно две еднакво големи части от матрицата. Ако повечето животни бъдат генерирани в дясната половина теоретично рутината, която отговаря за нея ще бъде много по-натоварена от другата и ще приключи след нея. Ала в случая всяка рутина итерира през всяка клетка от площта си и обработката на всяка клетка става за линейно време сложността й ще бъде length\*width. Дори а ако няма и едно животни в областта на рутината, изпълнението й отново ще бъде length\*width. Следователно броя животни няма да повлия значително на времето за завършване на рутината. Поради тази причина не е предприето динамично балансиране.

Комуникация между рутините

В първоначална реализация на алгоритъма, основните четири матрици бях зададени като глобални променливи и бяха използвани от всяка една рутина при обработката на ходовете на животните. Това предоставяше възможност за намаляване на комуникацията между отделните рутини по време на тяхната работа. За съжаление, съответния подход не предоставя ускорение на оригиналния алгоритъм с една рутина, а напротив - двойно забавя времето му за изпълнение при едни и същи размери на полето.

|  |  |
| --- | --- |
| Брой рутини | Време |
| 1 | 34.24798791 sec |
| 10 | 45.71019964 sec |

Въпреки че при направения опит комуникацията между обработващите рутини бе сведена до минимум, постоянния досег с глобалните масиви забавя процеса на обработка. Всяко ядро на процесора разполага със собствена кеш памет, която запазва често използваните ресурси в себе си, вместо да се изпълнява операцията за взимането им от главната памет. Така при процесор с n ядра една и съща информация може да бъде кеширана цели n пъти. Също така, ако кешираната информация се е променила, тя трябва в някакъв момент отново да бъде синхронизира и с главната памет, което още повече забавя достъпа до информация. Следователно рутини, изпълнявани върху различни ядра няма да могат да се възползват от кешираната информация на останалите и в конкретния случай общия масив ще бъде неколкократно кеширан.

Решението използвано за преодоляване на този проблем е всяка рутина да има локално копие на частта от масива, която обработва. За целта функцията, изпълняваща се като рутина, приема за параметър матрица със съответните по-малки размери. След приключване на работа, всеки работник връща като резултат на main рутината копие на своята локална променлива, която вече е обновена с нови положения на рибите.

Синхронизация на main и workers

За да може главната рутина да комбинира резултатите на всички останали, тя трябва да изчака всяка една от worker рутините да приключи. За целта се използват т.нар. Wait Groups в Golang. Те предоставят интерфейс на популярния проблем при семафорите – *barrier.* При него трябва множество процеси да се „срещнат“ в някакъв момент, т.е. всеки процес, който завърши с даден набор от работа, трябва да бъде приспан в този момент и да изчака всички останали да приключат също. В *Golang* , както беше споменато, това е реализирано чрез Wait Groups. Основна роля при тях играе един брояч, който представя броя на процесите, които още не са завършили работата си. Той може да бъде увеличаван с *Add и* намаляван *с Donе,* а За да се блокира процес се докато брояча стане 0, се използва функцията Wait. В случая преди стартирането на рутините инициализираме брояча с техния брой, а при завършване на изпълнението си, всяка рутина намаля този брояч с 1. Main рутината има поставен Wait след изпълнението на всички останали рутини, като по този начин тя трябва да изчака за резултат от всяка една от тях, за да завърши. [4]

Критична област

При този подход за разделяне на задачи трябва да се обърне внимание на крайните области обработвани на всяко подполе. По-конкретно колоните от клетки с най-голям и най-малък индекс в съответната подматрица. При конкурентна обработка на животните, намиращи се в тези колони е възможно да се получат колизии и несъвместимост с правилата, тъй като рибите и акулите тук могат да преминат на следващ ход към зона от матрицата обработвана от друга рутина. Ако това се случи, съседната рутина е възможно да се опита да постави в същия момент друг обект от нейната област в съответната клетка. Това би нарушило правилата на Wa-tor и би довело до подвеждащи резултати.

Критична област

Routine 5

Routine6

Routine4

Routine3 1

Routine2223 1

Routine1223 1

За справяне с критичната област е необходимо да се въведе комуникация между рутините. За тази цел въпреки забавянето използваме обща памет. Прилагаме метода CREW – Concurrent read, exclusive write. Създаваме глобален *map*, в който съхраняваме информацията за всяка клетка от четирите таблици. По този начин всички промени, които една рутина направи, ще могат да бъдат достъпни от всички останали. Така, ако Routine2 например премести риба в клетка на Routine3, съседната рутина ще е знае за промените.

За да се избегне race condition (едновременното редактиране на една клетка от две рутини) и да се постигне exclusive write, трябва да бъде ограничен достъпа до общия ресурс. За целта се използва мютекс, който забранява два процеса да имат едновременен достъп до map-a. Мютексът функционира като катинар за даден ресурс, за който съществува само един ключ. Мютексът дава този ключ на първия пристигнал процес и никой друг не може да достигне пазената памет, докато първия не върне ключа на мютекса. В проекта е използвана вградената библиотека на Golang за мютекс. Тя предоставя методите Lock и Unlock, които съответно отговарят на операцията взимане на ключ и освобождаването му. Рутина, когато достигне до обработката на критичната си зона, тя опитва да вземе ключ от мютекса с Lock и ако друга рутина в същия момент не работи върху своята критична зона, първата може да продължи. В противен случай тя изчаква да се освободи ключа. Очевидно това ще предизвика забавяне на изпълнението, но точно поради тази причина се използва map в реализацията. Като ключове на map-a са двойките от индекси на клетките от критичната област, а стойностите му са наредени четворки от данните в съответните матрици world, energy, starvation, breed. По този начин общият ресурс ще бъде заключен докато итерираме всички животни от критичната област, а не цялата критична област.

В таблицата отдолу може да се види сравнението на текущия проект с описаните по-рано реализации.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Num. | Модел | Балансиране | Технология | Синхронизация |
| 1. Overeinder, B. J., & Sloot, P. M. A. | MPMD (решетка) | Статично | C++ | Time Wrap + messaging |
| 2. Nicolo Tonci | MPMD (решетка) | Статично | C | Semaphores |
| 3. Personal project | MPMD (решетка) | Статично | Golang | Semaphores (WaitGroups and Mutexes) |

# Тестване на проекта

Характеристики на тестовата машина

Architecture: x86\_64

CPU op-mode(s): 32-bit, 64-bit

Byte Order: Little Endian

CPU(s): 32

On-line CPU(s) list: 0-31

Thread(s) per core: 2

Core(s) per socket: 8

Socket(s): 2

NUMA node(s): 2

Vendor ID: GenuineIntel

CPU family: 6

Model: 45

Model name: Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz

Stepping: 7

CPU MHz: 1394.470

CPU max MHz: 3000.0000



CPU min MHz: 1200.0000

BogoMIPS: 4389.52

Virtualization: VT-x

L1d cache: 32K

L1i cache: 32K

L2 cache: 256K

L3 cache: 20480K

NUMA node0 CPU(s): 0-7,16-23

NUMA node1 CPU(s): 8-15,24-31

Програмата се тества без допълнителни параметри и без визуализация. За отчитане на времето за тестване се използва стандартната библиотека на Golang “testing”, която предоставя функционалност за benchmarking.

Тестови сценарии

Ускорението на кода е тествано при два набора от променливи.

* **Length *=*** *490****,* width *=*** *980****,* numFish *=*** *1000****,* numSharks *=*** *200*
* **Length *=*** *980****,* width *=*** *1920****,* numFish *=*** *2000****,* numSharks *=*** *400*

Въвеждаме следните означения на променливите в изпълнението на кода:

* **p** – брой рутини
* **Tp1, Tp2, Tp3, Tp4** – времето за изпълнение в милисекунди на съответния опит
* **Tp** – средноаритметично от четирите опита при p на брой рутини
* **Sp**  - **T1/Tp** ускорението при p рутини
* **Ep – Sp/p** ефективността при p рутини

Тестови резултати

Сценарии 1

Таблицата показва резултатите получени на тестовия сървър с променливи **Length *=*** *490****,* width *=*** *980****,* numFish *=*** *1000****,* numSharks *=*** *200*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| p | Tp1 | Tp2 | Tp3 | Tp4 | Tp | Sp | Ep |
| 1 | 10747 | 11812 | 10473 | 10591 | 10905,75 | 1 | 1 |
| 2 | 5581 | 5387 | 5347 | 5759 | 5518,5 | 1,97621636 | 0,98810818 |
| 4 | 2706 | 2859 | 2645 | 2972 | 2795,5 | 3,90118047 | 0,97529512 |
| 8 | 2141 | 2187 | 1987 | 2039 | 2088,5 | 5,22180991 | 0,65272624 |
| 12 | 1284 | 1569 | 1358 | 1439 | 1412,5 | 7,72088496 | 0,64340708 |
| 16 | 1289 | 1361 | 1199 | 1253 | 1275,5 | 8,5501764 | 0,53438603 |
| 20 | 1289 | 1333 | 1173 | 1301 | 1274 | 8,56024333 | 0,42801217 |
| 24 | 1267 | 1405 | 1184 | 1152 | 1252 | 8,71066294 | 0,36294429 |
| 28 | 1224 | 1279 | 1232 | 1195 | 1232,5 | 8,8484787 | 0,3160171 |
| 32 | 1401 | 1413 | 1307 | 1215 | 1334 | 8,17522489 | 0,25547578 |

Графика на изменението на Тp при 4те опита

Графика на изменението на ускорението Sp

Сценарии 2

Таблицата показва резултатите получени на тестовия сървър с променливи **Length *=*** *980****,* width *=*** *1920****,* numFish *=*** *2000****,* numSharks *=*** *400*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| p | Tp1 | Tp2 | Tp3 | Tp4 | Tp | Sp | Ep |
| 1 | 42134 | 41501 | 41152 | 41914 | 41675,25 | 1 | 1 |
| 2 | 21005 | 21356 | 21190 | 21134 | 21171,25 | 1,9684832 | 0,9842416 |
| 4 | 13355 | 13263 | 13247 | 13453 | 13329,5 | 3,12654263 | 0,78163566 |
| 8 | 8467 | 8221 | 8392 | 8469 | 8387,25 | 4,96888134 | 0,62111017 |
| 12 | 5149 | 5168 | 5269 | 5164 | 5187,5 | 8,03378313 | 0,66948193 |
| 16 | 4352 | 4357 | 4338 | 4406 | 4363,25 | 9,55142382 | 0,59696399 |
| 20 | 4267 | 4276 | 4174 | 4278 | 4248,75 | 9,80882613 | 0,49044131 |
| 24 | 4003 | 3991 | 3972 | 3985 | 3987,75 | 10,4508181 | 0,43545076 |
| 28 | 4131 | 4045 | 4043 | 4045 | 4066 | 10,2496926 | 0,36606045 |
| 32 | 4125 | 4038 | 4160 | 4070 | 4098,25 | 10,1690356 | 0,31778236 |

Рутините са по-малки от нишките, но въпреки това може да се изпълнява по една рутина на нишка. Го Виждаме, че най-бурно е ускорението до 16 рутини. Това най-вероятно се дължи на това, че процесорът е hyper-threaded, т.е. всяко ядро разполага с две нишки. Следователно реалните ядра са 16, а не 32 и поради тази причина не наблюдаваме толкова голямо ускорение след 16 рутини.

# Литература

[1] Alexander Keewatin Dewdney, Sharks and fish wage an ecological war on the toroidal planet wa-tor, *Scientific American, pp. 14-22, 1948.*

[2] Overeinder, B. J., & Sloot, P. M. A. (1993). Application of time warp to parallel simulations with asynchronous cellular automata. In A. Verbraeck, & E. J. H. Kerkchoffs (Eds.), European simulation symposium 1993, pp. 397-402

[3] Nicolo Tonci, Wator Progetto di Sistemi Operativi, 2015. (<https://github.com/nicolotonci/ParallelWator/blob/master/relazione.pdf> )

[4] Allen. B. Downey, The little book of semaphores, pp. 12-15, 2009

# 