Изследване на скалируемостта на Wa-Tor симулацията при статично и динамично балансиране

Проект по “Системи за паралелна обработка”

Изготвил:
Антонио Ерол Милев
ФН: 82074, Група: 5, Курс: 3

Съдържание

1. Увод

Wa-Tor[2] е локално синхронен алгоритъм при паралелното програмиране. Представлява симулация на двумерен свят с формата на тор, в който наблюдаваме популацията на два вида - херинги и акули.

Основата на паралелността ще се изразява в декомпозирането на домейна. Идеята е че разбиваме домейна (двумерния масив на тора) на няколко поддомейна, които ще бъдат обработвани паралелно от различни процеси. В случая на този проект разбиваме решетка на няколко реда. Тъй като Wa-Tor е локално синхронен ще се наложи синхронизация между граничните зони на поддомейните, тъй като рибите и акулите могат да преминават от един поддомейн в друг. Вътрешността на всеки поддомейн си се изчисля за себе си обаче и това ще доведе до ускорение на програмата. Има два вида декомпозиция - статична и динамична. При статичната разбиването се прави веднъж в началото и повече не се променя. Тъй като има случаи, в които работата на единия процес може да бъде много по натоварена от друга и да трябва да се изчакват понякога е удачно да се динамично декомпозиране. При него поддомейните могат да се променят и в движение за да компенсират тези разлики. Това обаче внася свъртовар и не винаги е удачно за използване.

В този проект решението използва статично декомпозиране. В началото бях планирал като бонус да има и елементарен вариант на динамично декомпозиране[4] за да бъде направено сравнение дали то предоставя достатъчно ускорение за сметка на свръхтовара. За съжаление няма да мога да имам възможност да работя по проекта през следващата седмица, така че за момента оставам без него.

2. Wa-Tor правила

Wa-Tor е симулация тип клетъчен автомат (Cellular Automata). Светът представлява тор, който се моделира чрез обикновен правоъгълник, чиито страни са свързани една с друга. Тоест ако една клетка излезе от десния ръб ще се появи от левия, ако излезе от долния ще се появи от горния и обратно. Светът е разделен на квадрати (двумерна матрица). Всеки квадрат е една клетка, която може да бъде празна, херинга или акула. Времето ще тече на дискретни итерации. Всяка итерация всяка непразна клетка бива обработена с приоритет от дясно на ляво и от горе на долу. Точните правила, по които се обработват клетките са следните:

Херинги:

- Придвижва се на случайно избрана пряка съседна позиция (без диагонално движение). Ако няма такава остава на едно място.
- Всяка херинга има време за репликиране. След определен брой итерации при следващото си преместване херингата ще се размножи и остави на нейното място нова херинга. В такъв случай времето за репликиране се нулира.

Акули:

- Придвижва се на случайна избрана пряка съседна позиция, в която има херинга и изяжда херингата в процеса. Ако няма такава се придвижва на произволна празна или остава на място ако няма и празна.
- Акулата има време за репликиране подобно на херингата и се размножава по същия начин.
- Освен това им и време на гладна смърт. Ако изминат достатъчно на брой итерации без акулата да изяде херинга тя умира. Това време се нулира когато акулата изяде някоя херинга.

3. Употреба на проекта

Кодът на проекта може да бъде намерем в github на този линк- . Проектът зависи от две C++ библиотеки - *SFML/Graphics[1]* и *boost/program_options.* След като те бъдат инсталирани кода може да бъде компилиран със следните две команди:

```
$ g++ -std=c++17 -c wator.cpp
$ g++ -std=c++17 wator.o -o wator-app -lsfml-graphics -lsfml-window
-lsfml-system -lboost_program_options -pthread
```

След като бъде компилиран, бинарния файл може да бъде стартиран с множество опции от командния ред. За всяка от тях има и стойност по подразбиране, за да може да симулацията да бъде пускана бързо и без да трябва да се напише всяка една от тях. Пълния списък на опции може да бъде видян с флага *--help.*

```
antonio@antonio-B85M-D2V:~/fmi-dev/spo$ ./wator-app --help
Allowed options:
  --help                              Show help description
  -c [ --chronons ] arg (=300)        Set simulation chronons (iterations).
  -w [ --width ] arg (=2000)          Set simulation width in pixels.
  -h [ --height ] arg (=1000)         Set simulation height in pixels.
  -z [ --cell-size ] arg (=5)         Set square cell size in pixels. Must
                                      divide both width and height of
                                      simulation.
  -t [ --threads ] arg (=1)           Set number of threads
  -f [ --fishes ] arg (=5000)         Set number of fishes
  -s [ --sharks ] arg (=1000)         Set number of sharks
  -F [ --fish-replicate ] arg (=10) Set fish replicate time in chronons
  -S [ --shark-replicate ] arg (=8) Set shark replicate time in chronons
  -R [ --shark-starve ] arg (=7)    Set shark starve time in chronons
  --dry-run                           Run the simulation without visualization
                                      and output.
  --make-gif                          Produce a gif animation
```

За тестовите цели ще бъдат използвани главно флаговете *--width --height* за размер на симулацията както и *--threads* за броя нишки. След приключване на

симулацията, ако не е бил използван *--dry-run* флага, ще бъде създаден .gif в работната директория с име ‘wator.gif’, който показва генерираната анимация.

4. Описание на паралелния алгоритъм

Паралелният ни алгоритъм работи основно на принципа на статичната декомпозиция. В началото на програмата екрана бива разбит на N на брой колони, по една за всяка нишка. Заради начина на визуализация, визуално това биха били колони, но вътрешно в кода са по-скоро редове. Това на практика ще бъдат поддомейните, с които ще работим. Редовете са с еднаква дебелина за да може техния товар да се балансира. Ако размера на екрана не се дели точно на броя нишки то остататък се добавя към последната нишка. Използвайки редове квадрати тип шахматна дъска минимизираме границите между поддомейните и времето за комуникация между тях, което съответно намаля свърхтовата както и сложността на алгоритъма.

Алгоритъмът, по който работят нишките, е следният. Всяка една от тях започва да работи паралелното с останалите започвайки от най-горния си ред надолу спазвайки вече описаните правила на Wa-Tor. При обработка на първи ред всяка нишка комуникира с нейния предишен съсед, че е приключила работата си по него. Междинните редове обработва без да трябва да прави допълнителна работа. Когато някоя от нишките стигне последния си ред тя трябва да изчака следващият си съсед да приключи обработката на неговия си първи ред. По този начин си гарантираме, че няма нишките няма да пипат по една и съща памет едновременно. Това би довело до непредсказуеми резултати. Една малка забележка, е че така се нарушава правилото в какъв ред се обработват клетките на двумерната матрица. Например може херинга от долния съсед да е навлязла в горния. При изчакване за горния съсед вече има една клетка заета, която може би той сам щеше да си заеме. Тъй като така или иначе разчитаме на генератор на случайни случай приемаме, че това не е фаталано. Ако трябваше горния съсед да си заеме обратно заетата клетка това значи долния да преизчисли нова позиция за рибата. Но това съответно значи, че тя може да иска да заеме съседна позиция заета от друга риба, над която по принцип би имала приоритет. И така може проверката може да се връща назад

безкрайно. Фигурата по долу демонстрира подобна ситуация. Виждаме, че клетка 1 е преминала граница и е навлезнала в региона на горната нишка. Сега ако клетка две избере да застане на мястото 1 тя би трябвало да е с приоритет (клетка 2 нарочно е по далече от 1 за да има място за зачеркната стрелка). Ако клетка 1 трябва да се измести в друга позиция може случайно да избере мозицията на 3, на което би трябвало да има право защото е с приоритет пред 3 защото е по-надясно. Следователно трябва да преместим и 3. Така може да продължим и към 4 и тн. Затова просто ще игнорираме този проблем.

Синхронизацията между нишките се прави с помощта на семафори. Използват се масив от семафори - *mutexes*, с дължина равна на броя нишки. По-долу е представен псевдокод на синхронизацията между нишките. Нека в него [*startRow*, *EndRow*) е интервалът, в който дадената нишка работи. *threadIndex* е номера на нишката а *threadsCount* общия брой нишки.

```
Lock(mutexes[threadIndex]);
CalculateRow(startRow);
Unlock(mutexes[threadIndex]);

for i := startRow + 1; i < endRow - 1; i++ do
    CalculateRow(i);
```

end for

Lock(*mutexes*[*threadIndex* + 1 **mod** *threadsCount*]);
CalculateRow(*endRow* - 1);
Unlock(*mutexes*[*threadIndex* + 1 **mod** *threadsCount*]);

След като всяка нишка обработи своята зони, те отново се изчакват и чак тогава се започва следващата итерация на симулацията. Това на практика означава че нашият алгоритъм е бърз колкото най-бавната нишка.

5. Софтуерна Архитектура

Проекта е написан на C++ като се използват библиотеки, за които е нужен стандард C++17 или по-нов. Библиотеки и класове, които използвам:

- **std::boost::program_options** - за обработка на аргументи и опции от командния ред
- **SFML/Graphics** - за визуализация на симулацията
- **std::thread [5]** - за работа с нишки
- **std::mutex** - за синхронизация между нишките

Освен тях има написани три класа от мен: **Cell**, **Fish** и **Shark**. Fish и Shark наследят абстрактният клас Cell. За представяне на света използвам двумерен вектор от Cell показатели. Обяснено с думи програмата работи по следния начин. Започва като обработва командните аргументи и задава параметрите на симулацията. Правят се и проверки и затова дали те са валидни. Например не може да бъдат зададени повече риби и акули отколкото размера на света би побрал. След това двумерния вектор от Cell показатели се попълва на случаен принцип спазвайки зададените параметри. След това вече започва паралелната част. За всяка нишка се създава нов std::thread и той започва работа си по вече описания алгоритъм, синхронизирайки се чрез std::mutex. След приключване на една итерация на симулацията и в случай, че програмата не е пусната в режим --dry-run нишките рендерират върху екрана новото състояние. Ако имаме и опция --make-gif то се запазва и снимка.

Кода може да бъде разгледам подробно в гитхъб -

6. Анализ

Тестова машина:

- Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz, 8 ядра с по 2 нишки, 32 логически чрез Hyperthreading[3]
- 32K L1 кешове и RAM - 64 GB

С тези характеристики можем да очакваме ускорение само до 32 нишки като очакваме постепенно то да намаля. Особено след 16тата очакваме значително по лоши резултати тъй като 32 логически ядра с Hyperthreading не биха могли да достигнат ефективността на 32 реални ядра.

Времето на алгоритъма се измерва от започване на работа на нишките, тоест прескачаме началното инициализиране на света. Също така засичаме в режим без визуализация и запазване на изображения. Направени са измервания за различни на брой нишки от 1 до 32 по 3 теста за всеки N на брой нишки. От тези 3 теста се взима най-бързия за да е сигурно, че не е станало забавяване от “шум” в машината.

Грануларността, която съм тествал е единствено едра грануларност g = 1. Тъй като алгоритъмът е локално синхронен не изискваме да се тества по голяма грануларността. Дори тя да носи някакво ускорение повечето пъти не си заслужава свръхтовара. Особено в Wa-Tor алгоритъма състоянието на света се мени много бързо. Това означава, че ако направим балансиране по средата на работа, то ще трае съвсем малко време преди да вече товара да е напълно различен.

7. Тестови резултати

Изследвал съм скалируемостта при 2 различни опции на програмата

- Width = 2000, height = 1000, chronons = 100, sharks = 5000, fish = 50000

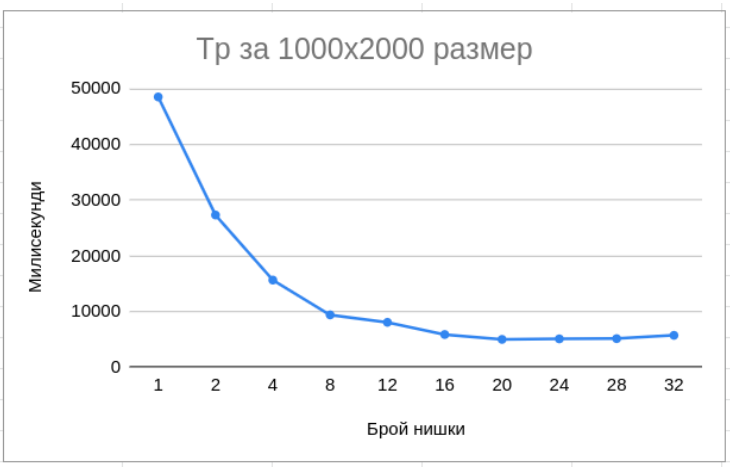
- Width = 4000, height = 2000, chronons = 100, sharks = 100000, fish = 1000000

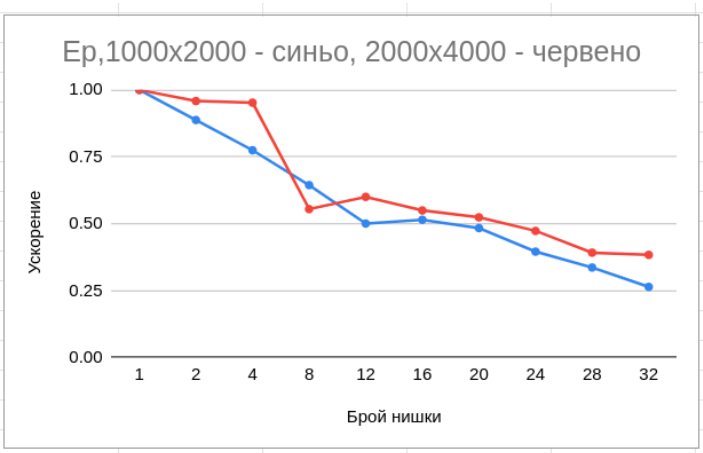
В долните таблици и графики са описани резултати от тестване. Значение на обозначенията. Времената са в милисекунди

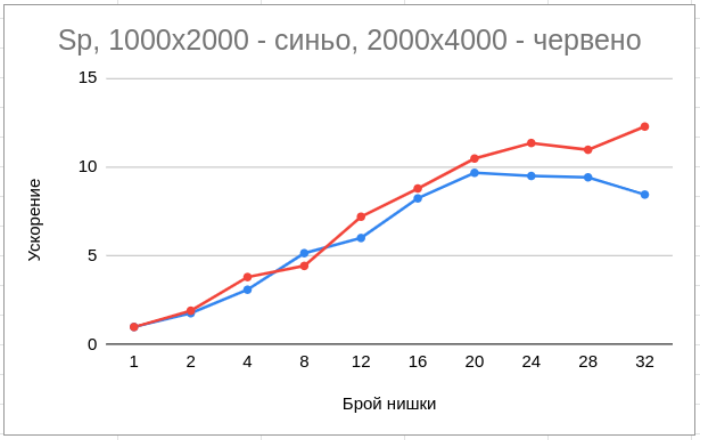
- Tp n - време за изпълнение на n-тия тест за p нишки
- Tp - min(Tp n)
- Sp = T1 / Tp
- Ep = Sp / p

1000x2000 размер, обща начална популация 55000							
Тест номер	Брой нишки, p	Tp 1	Tp 2	Tp 3	Tp min	Sp	Ep
1	1	49690	48521	49591	48521	1	1
2	2	28184	27401	27337	27337	1.774920438	0.8874602188
3	4	16385	15663	15789	15663	3.097810126	0.7744525314
4	8	9748	9415	10049	9415	5.153584705	0.6441980882
5	12	8465	8077	8208	8077	6.007304692	0.5006087244
6	16	5978	5892	6203	5892	8.235064494	0.5146915309
7	20	5165	5396	5016	5016	9.673245614	0.4836622807
8	24	5346	5375	5109	5109	9.497161871	0.395715078
9	28	5595	5869	5151	5151	9.419724325	0.3364187259
10	32	6197	5976	5743	5743	8.448720181	0.2640225057

2000x4000 размер, обща начална популация 1100000							
Тест номер	Брой нишки, p	Tp 1	Tp 2	Tp 3	Tp min	Sp	Ep
1	1	144065	150554	144911	144065	1	1
2	2	76908	78360	75165	75165	1.916650037	0.9583250183
3	4	39479	37820	38166	37820	3.809227922	0.9523069804
4	8	32472	33087	32931	32472	4.436591525	0.5545739406
5	12	20385	19997	20215	19997	7.20433065	0.6003608875
6	16	16385	17109	16763	16385	8.792493134	0.5495308209
7	20	13928	13750	13946	13750	10.47745455	0.5238727273
8	24	12854	12840	12691	12691	11.35174533	0.4729893888
9	28	13174	13333	13132	13132	10.97053	0.391804643
10	32	11957	12516	11736	11736	12.27547716	0.3836086614



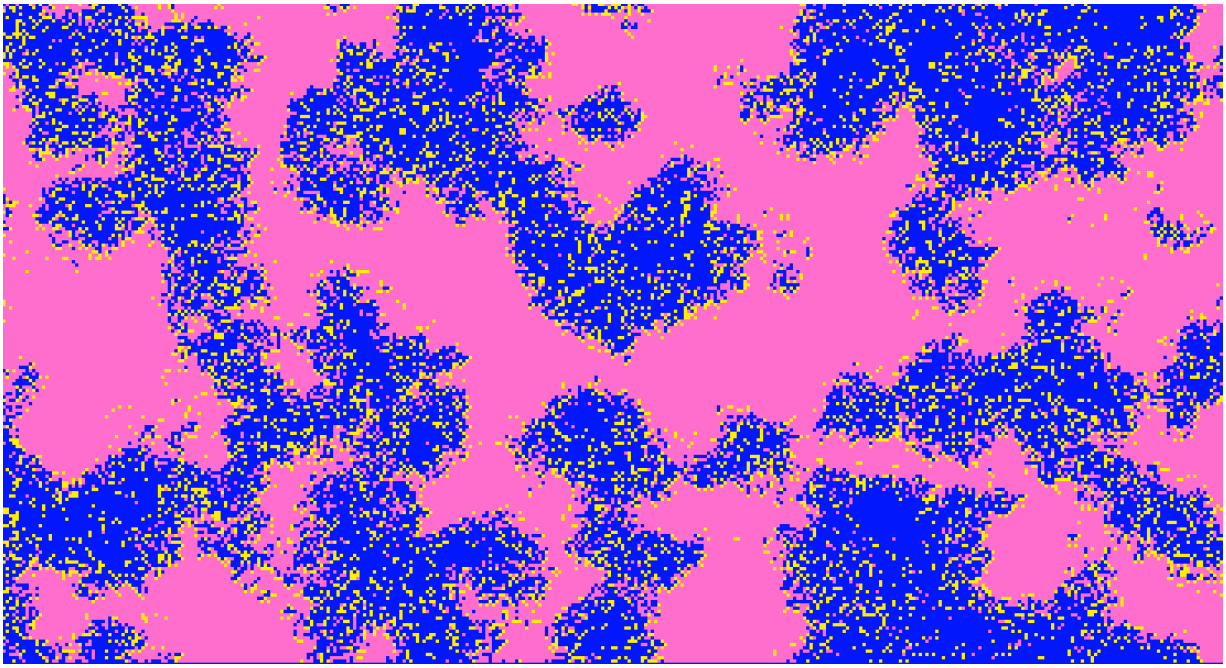




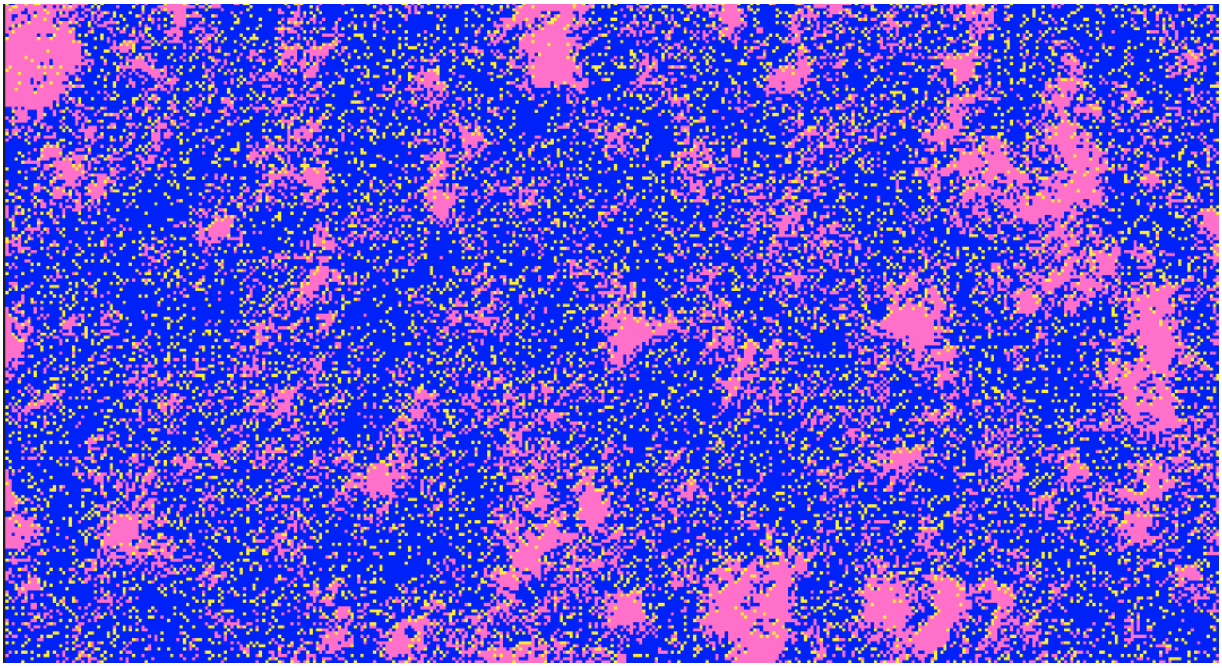
8. Визуализация на симулацията

Освен обикновенната симулация, която просто изкарва резултат за изпълнението и, към проекта има и визуализация на света направена използвайки SFML библиотеката за C++. Има вариант както за показване в реално време така и за запазване на .gif файл със симулацията. Това може да служи както за дебъг така и просто за интересен начин на демонстрация.

На долните екрани могат да се видят кадри от визуализацията. Херингите в розово а акулите в жълто.



Размер 1000x2000 с размер на клетка 5 пиксела. Наблюдаваме много размножили се херинги, което дава възможност на акалуте да не умират и постенно изяждат популацията херинги.



Размер 2000x4000 с размер на клетка 5 пиксела. Акулите са много повече от херингите, което води до почти изчезване на популацията херинги, а от там и изчезване на акулите. Резултата е или изчезват и двата или херингите завземат всичко след измирането на всички акули.

9. Бъдещо развитие

Проектът е далеч от идеален и може да бъдат направени множество подобрения. Оптимизации както на самия алгоритъм на симулацията така и на паралелността. Например може да се добави по-бързо обхождане чрез списък от клетките, в които има нещо и да се пропуснат множество празни проверки. Паралелността може да се подобри също, особено при рендерирането. Може да се направи по-умно динамично разбиване както и допълнителни изследвания с различна грануларност. Не на последно място кодът може да бъде рефакториран тъй като в момента е само един не много добре подреден файл.

Литература

[1] Milcho G. Milchev, SFML Essentials
https://www.packtpub.com/product/sfml-essentials/9781784397326

[2] Alexander Keewatin Dewdney. Sharks and fish wage an ecological war on the toroidal planet wa-tor. Scientific American. pp. I4—22, 1984

[3] Jon Stokes. Introduction to multithreading, superthreading and hyperthreading. Ars Technica. pp. 2–3. Retrieved 30 September 2015., 2002.

[4] F. Baiardi, A. Bonotti, L. Ferrucci, L. Ricci, and P. Mori. Load balancing by domain decomposition: the bounded neighbour approach. In In Proc. of 17th European Simulation Multiconference, pages 9–11, 2003.

[5] https://en.cppreference.com/w/cpp/thread