Hell-Body

Изследване на скалируемостта на броя тела на n-body при употребата на ортогонално рекурсивно разделяне и алгоритъма Barnes-Hut

Изготвил: Николай Кормушев, ФН: 81805, Курс: 3, Група: 1

Съдържание

1. Увод

1.1 История и идея на задачата

В физиката проблемът n-body се отнася до предсказването на движенията, предизвикани от гравитационните взаимодействия между различни множества от астрономически обекти. Буквата n от името на проблема се отнася до броя тела, които наблюдаваме. Задачата се е зародила през 17-ти век. Целта е била по-добре да си обясним движенията на астрономическите обекти от нашата слънчева система. Проблемът първоначално е зададен от Исак Нютон, който установил, че е недостатъчно да знаем началната позиция и скоростта на обектите, за да изчислим техните взаимодействия, а ни трябва информация за силите на гравитационно взаимодействие. Това довело до много затруднения за решаването на задачата. Така продължило до 19 век, когато крал Оскар II на Швеция обявил награда за всеки, който реши задачата успешно. Това довело до успешното ѝ решаване при взаимодействието между 2 тела от Poincare и между 3 от Karl Fritof.

1.2 Математиката зад n-body

Нека имаме n тела, всяко с маса m_i и позиция q_i в $\mathbb{R}^3$, които се под влиянието на гравитационни привличания за $i \in \{1, \dots, n\}$. Ще използваме, че $F = m * a$. Това означава, че $m_i * \frac{d^2 qi}{dt^2} = m_i * a_i$ е равно на сумата от силите, които действат върху тялото с индекс i. Закона на Нютон за гравитацията, казва, че гравитационната сила, която тялото i усеща от тялото j се дава от уравнението $F_{ij} = G * \frac{m_i * m_j}{\|q_j - q_i\|^2} * \frac{(q_j - q_i)}{\|q_j - q_i\|}$, където $\|q_j - q_i\|$ е разстоянието между телата, а G е гравитационната константа. Тогава имаме, че $m_i * \frac{d^2 qi}{dt^2} = \sum_{j=1, j \neq i}^{n} G * \frac{m_i * m_j}{\|q_j - q_i\|^2} * \frac{(q_j - q_i)}{\|q_j - q_i\|}$. Понеже знаем масите m_i и координатите q_i, $i \in \{1, \dots, n\}$ значи можем да получим ускорението a_i, а от там и да сметнем и новата скорост на тялото след време Δt, равна на $v_{i\text{ново}} = v_{i\text{старо}} + a_i * \Delta t$. Сега, използвайки скоростта можем да обновим позициите на обекта i по следния начин $q_{i\text{ново}} = q_{i\text{старо}} + v_{i\text{ново}} * \Delta t$. Използвайки този метод можем да изчислим точно движенията на телата в за една итерация от време Δt. Повтаряйки тези сметки, можем успешно да симулираме движението за произволни количества от време. Алгоритъмът, който ще имплементираме ще счита, че пространството е двуизмерно, което според [1] означава, че формулата за силите има вида $F_{ij} = G * \frac{m_i * m_j}{\|q_j - q_i\|} * \frac{(q_j - q_i)}{\|q_j - q_i\|}$.

2. Анализ на n-body

2.1 Сравнение на различните алгоритми за n-body [2]

2.1.1 Увод:

Първата фаза от проучването ми за n-body се състоеше в това да проуча какви са различните алгоритми, които мога да имплементирам и до какво ускорение ще ме доведат. Най-наивният алгоритъм се състоеше в това да разделя телата между процесите и да смятам взаимодействията

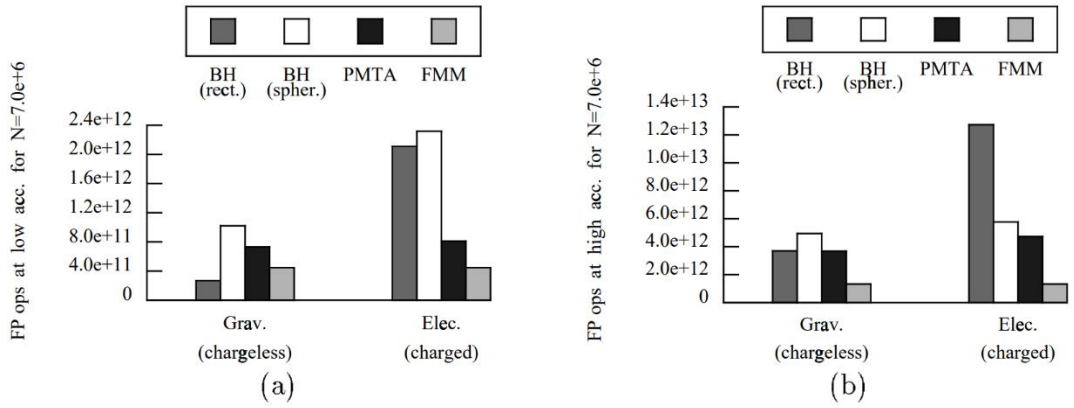
между всеки и две. Сложността му е $O(n^2)$. Очакването ми беше, че трябва да има по-бързи и интересни алгоритми от него и задълбах и намерих [2].

2.1.2 Идея на дървовидните n-body алгоритми [2]:

В този труд се сравняват 3 алгоритъма Barnes-Hut(BH), Fast multipole method(FMM) или преведено бърз многополюсен метод, както и Parallel Multipole Tree Algorithm(PMTA), което преведено е паралелен многополюсен дървесен алгоритъм. Тези алгоритми спадат в категорията алгоритми базирани на дървета. Те има сложности съответно $O(n \log n)$, $O(n)$ и $O(n)$, ако ги разглеждаме като последователни алгоритми. И трите алгоритъма правят някакъв вид апроксимации. Първият счита, че ако едно тяло е достатъчно далеч от едно множество от тела, то множеството може да се счита за едно тяло с координати центъра на масата на телата в множеството и маса сумата от масите. Така смятаме едно взаимодействие вместо по едно за всяко тяло от множеството, което тривиално ни намаля сметките. Този алгоритъм в анализът им е бил имплементиран по два начина. Един път със сферична и един път с праволинейна координатна система. Вторият алгоритъм изчислява многополюсни експанзии и използва многополюсни моменти. Не съм запознат с физиката зад тях и не разбирам съвсем точно как работят, за да мога да го имплементирам и това до голяма степен ме направи по-склонен да пробвам първия. Голяма част от научните трудове ползват този метод и или поне многополюсни експанзии в комбинация с BH. При моята имплементация аз се стараех да се придържам само към физиката, която разбирам и обясних в 1.2. PMTA е комбинация на BH и FMM. Използва многополюсни експанзии и моменти, но същевременно има идеята, че тялото трябва да се намира достатъчно далеч, за да ги използваме. Пак се гледа дистанцията между тяло и множество от тела, за да се прецени дали да се ползват апроксимации. Този алгоритъм също ми беше неясен на база липсата ми на познания по физика. Все пак прочетох до какви заключения са стигнали, че ми беше интересно, макар и да беше ясно, че нямам друг избор освен да си пробвам късмета, като реализирам Barnes-Hut.

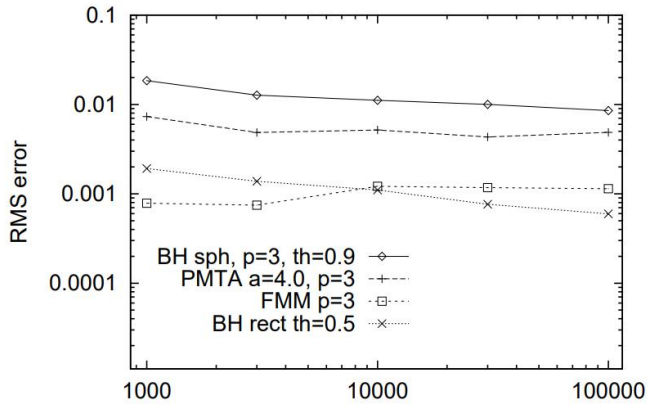
2.1.3 Анализ на алгоритмите от [2]

За метрика е бил използван броя на взаимодействията при различните алгоритми. Установили са, че при праволинейни координатни системи и стандартно разпределение и размерност 10^8 и нагоре BH се справя най-добре. При сферични или някакъв друг вид дистрибуции, като електростатични другите алгоритми са имали по-малко на брой взаимодействия. Отдолу се виждат разликите в броя на взаимодействията:



Фигура 1. Сравнение на брой изчислени взаимодействия по алгоритъм

На фигурата се виждат броя взаимодействия при ниска точност а) на апроксимациите , а в б) при висока. Ясно се вижда, че FMM при електростатична дистрибуция и висока точност е най-бързо, а BH при ниска точност и гравитационна дистрибуция. Също така в [2] са разгледани и грешките при апроксимациите



Фигура 2: сравнение на грешката

Ясно се вижда, че при праволинейна имплементация на BH имаме най-малка грешка, което е плюс за алгоритъма, който ще реализирам.

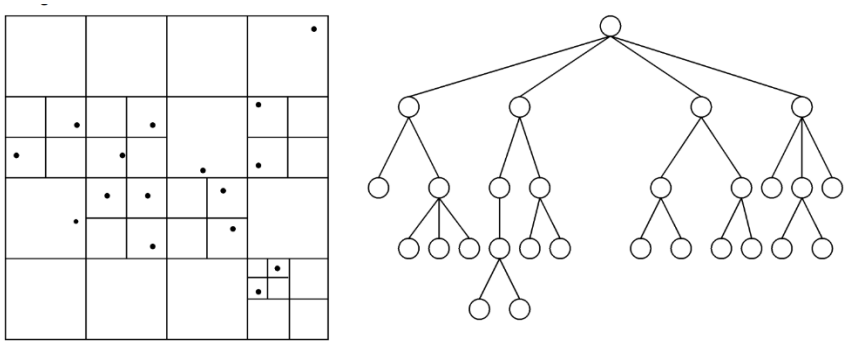
2.1.4 Заключение и избора на Barnes-Hut

Като цяло всеки алгоритъм си има ползи, а най-голямата за мен е лесната имплементация и разбираемостта. В случая алгоритъма, който имам най-голям шанс да реализирам е именно Barnes Hut, а както и се вижда от анализа и неговото ускорение може да е достатъчно бързо особено за нашите цели.

2.2 Имплементации на Barnes-Hut [3],[4]

2.2.1 Как работи последователния Barnes-Hut

Преди да продължа с паралелната имплементация ми се стори добра идея да обясня точно как работи Barnes-hut алгоритъма последователно. Идеята му е както по-горе обясних да правим апроксимации за множества от тела, които са много далеч от нас, но как точно правим това? За целта взимаме света и го разделяме на 4. Гледаме дали във всяка клетка има под L тела. Ако да, спираме. Ако не клетките с повече от L тела пак ги делим на 4 и така нататък. Това задава дърво, на което всеки възел има по 4 деца освен листата, в които има до L тела от света. После при самите изчисления за всяко тяло обхождаме дървото и гледаме дали центърът на маста на телата под възела е достатъчно далеч от нашето тяло. Ако е правим апроксимация с координата центъра на масата и мас сумата от масите на телата и прилагаме сметките от 1.2. В противен случай влизаме в децата на възела, ако е междинен и смятаме правим същата проверка за тях. Ако стигнем до листо смятаме взаимодействията директно с всеки елемент от листата. Ето примерна диаграма на BH дърво при L = 1 и съответстващия му свят как е разбит взета от [4]:



Фигура 3: Barnes-Hut дърво

Горе света първо е бил разбит на 4 равни парчета, които са 4 деца на корена. После те са били разбити и техните деца с над 1 тяло също и т.н. После във всеки сектор е имало над едно тяло и те са били разбити докато не е останало по едно тяло в листо/квадратче. В диаграмата не са показани възлите без тела в тях, които съответстват на празните квадратчета, но поне в моята имплементация и тях ги има.

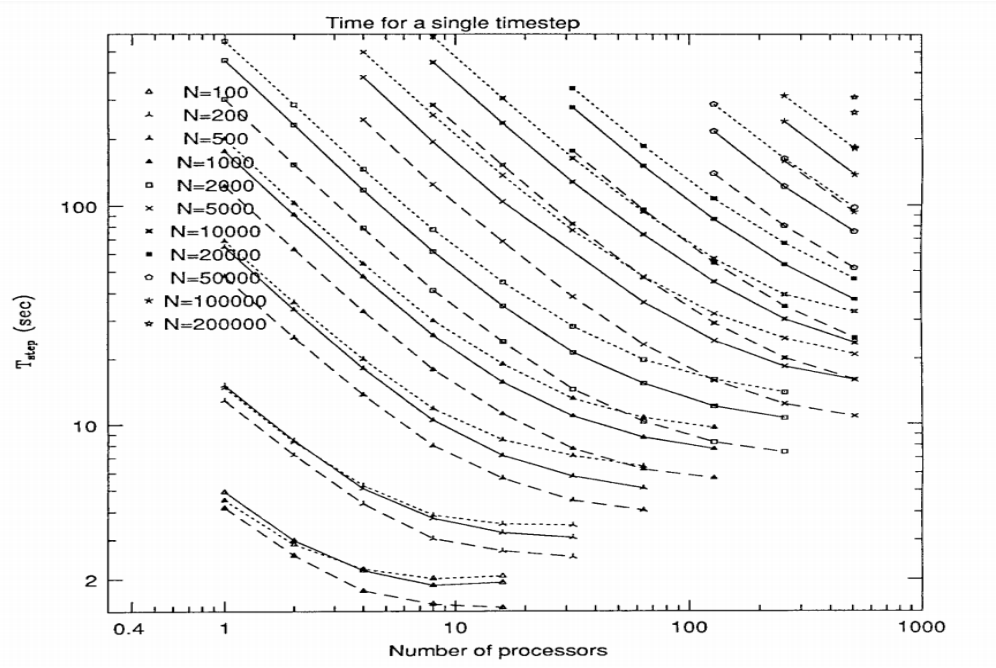
2.2.2 Идея на тезисът на J. K. Salmon [3]

Това може би е труда, от който най-много извлякох като информация и на който се основава моя проект. Алгоритъмът е от тип SPMD. При него не се използва споделена памет, а се предават съобщения(message passing). Salmon е един от първите, които са използвали ортогонално рекурсивно разпределение (ORB) заедно с BH дървета и са го реализирали паралелно. За този алгоритъм се строи така нареченото ORB дърво. Идеята е, че света се дели на сектори от тела, на които като им сумираме теглата ще получим едно и също число. Тези сектори съдържат телата за всеки процесор. Едно основно нещо, което е въвел Salmon е метода за комуникация между процесорите, който следва модела на хиперкуба, за да си обменят тела. Идеята на алгоритъма се

основава на това, че всяко тяло ще си построи локално BH дърво, но това дърво ще се състои само от телата, които са му жизнено важни, а ако не му трябват на тяхно място се слагат апроксимации, които репрезентират множество от тела, като координатите им са центърът на масата и масата е сумата от масите и дървото под тях в общия случай не е построено. Така всеки процесор има нужната информация да изчисли силите за своите тела и има много по-ниски BH дървета. BH дървото и ORB дървото при тази имплементация се строят на всяка итерация наново. Това не е проблем, защото те се строят бързо и в сравнение с изчисленията не заемат почти никакво време. Тук ще уточня и че Salmon за разлика от мен е използвал многополюсна експанзия и е изчислявал многополюсни моменти за апроксимациите. Тази част както преди не ми беше съвсем ясна и основно се фокусирах върху идеите на паралелната имплементация, като сметките ги направих както в 1.2. В повече детайли за идеите на Salmon ще навляза при имплементацията, защото иначе много ще се припокрива информацията. Затова тук ще се фокусирам основно върху ускорението и резултатите, които е получил.

2.2.3 Постигнати резултати от Salmon [3]

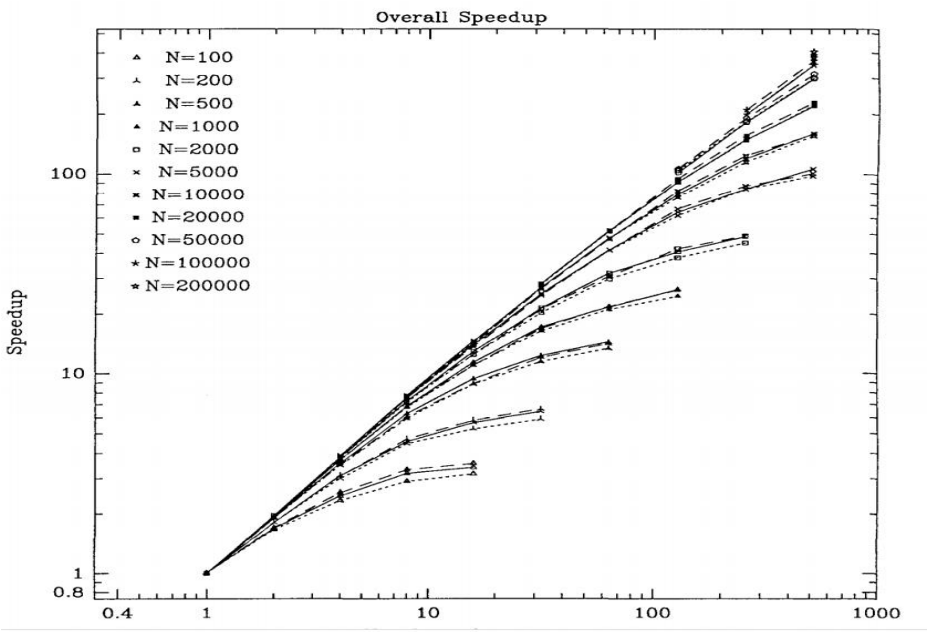
Машината използвана от Salmon при анализът му е с 512 процесорна NCube система, която е най-голямата система, до която е могъл да се добере и за тогавашните време едвам е покривала изискванията за суперкомпютър според това, което е написано. Системата е разпределена, взимайки предвид имплементацията, използваща обмен на съобщения. Постигнатите резултати за една стъпка се виждат в следната фигура:



Фигура 4: Времето за една итерация при Salmon

От диаграмата ясно се вижда, че е постигнал не малко забързване във времето за една итерация. Особено при по-големи количества данни от сорта на 200000. Забавяне е имало чак при брой процесори наближаващ 100. Също така може да се забележи, че броя процеси, за които е тествал са степен на двойката, което ще обясня по-късно и го има и при мен. Алгоритъмът не работи, ако

броя процеси не изпълнява това изискване. Тук вече можем да видим и ускорението, което при достатъчен брой тела изглежда почти линейно:



Фигура 5: Ускорението при Salmon

2.3 Преживявания с Паралелна n-body симулация [4]

Този научен труд, както моя се базира на Salmon. Той разширява някои от идеите там и ги оптимизира, така че да работят по-бързо. Иначе пак е SPMD и пак е с обмен на съобщения. Основните оптимизации са това, че вече ORB дървото не се строи на всяка итерация. Проверява се дали има повече от 5 процента разлика между разпределението работата на процесорите през последната итерация и ако е имало тогава се строи наново, което аз също правя. Също така вместо цялото дърво да се строи наново се строи само частта от него, в която се е нарушил баланса. Аз тук просто строя дървото наново изцяло. При BH дървото от друга страна вече се строи глобално дърво и то се балансира на всяка итерация спрямо локалните. Това глобално дърво помага и за това да не се налага да се строят постоянно локални дървета, а и те просто се модифицират при

нужда. Друга оптимизация е, че при строенето на локални дървета вече за телата се гледа в нещо като радиус около тялото дали другите процеси имат нужда от него и проверяваш, ако има процес достатъчно близо до него с важни тела се праща. Ако няма се очаква, че родителя ще му трябва на съседа и се праща той. Последната оптимизация е, че са заделили парче кеш, в което се съдържат жизнено важните части за изчисляване на телата, което води до голямо забързване при тях. Голяма част от тези оптимизации не съм ги правил. Взел съм част, ако съм мислел, че мога лесно да ги добавя, но нямах времето да имплементирам всички.

2.4 Анализ на [4]

Машината, на която са си провеждали тестовете е Connection Machine CM-5, която е с 256 възела за обработка на данни всеки, от които е с 128 мегабайта памет и извършва дробни операции със скорост от 160 Mflop/s. Тук няма толкова да задълбавам в анализа. Ще използвам обаче таблицата на фиг. 12 от [4], която сравнява имплементацията на Salmon с тяхната имплементация по-скорост и време за изчисления. Те са пускали симулацията с 256 процеса. Това ще ми е малко непосилно и затова ще пробвам да я пусна с колкото мога в случая 32 и да запиша моите резултати. Ако считаме, че при удвояване на процесите времето за работа намаля двойно се вижда, че моите резултати са съвсем прилични в сравнение с WS92 и WS93. [4] обаче би работило по-бързо.

	WS92	WS93	[4]	Hell-body
Машина	512-Delta	512-Delta	256-CM-5E	32-intel
# тела($x10^6$)	8.8	8.8	10	8.8
Разпределение	Равномерно	Равномерно	Равномерно	Равномерно
Време за изчисление на една итерация	77 секунди	114 секунди	26 секунди	336 секунди
Време за изчисления	85%	47%	76%	97%
Останало време	15%	53%	24%	3%

Фигура 6: Таблица сравняваща проучените алгоритми. WS означава Warren Salmon. WS92 за разлика от WS93 във времето за изчисления включва и обхождането на дърветата

2.5 Използвани технологии

Алгоритъмът го реализирах на езика C++. Доводът ми беше, че с помощта на език от по-ниско ниво мога по-добре да си оптимизирам кода. Също така това е езикът използван в [4]. Аз също, както в източниците си използвах обмен на съобщения и съответно библиотеката OpenMPI, където MPI стои за Message Passing Interface. Реших да пробвам тази имплементация, защото основно за това бях чел и беше започнало да ми се сформира идея как би се реализирало.

За генериране на графиките и GIF-овете съм използвал езика R заедно с библиотеките ggplot и gganimate, които са доста интуитивни за ползване, макар че точат доста от RAM-та на машината ми.

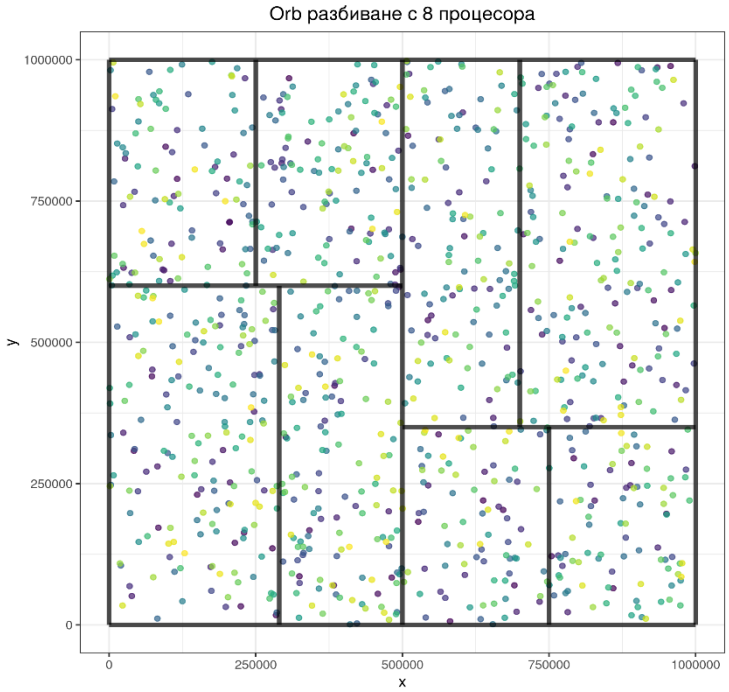
3. Hell-body имплементация

3.1 Разпределение на данните

3.1.1 Наивно разпределение

При паралелната реализация на Barnes-Hut алгоритъма на пръв поглед нищо не ни пречи да разпределим телата по равно между всички процесори. Това обаче води до дисбаланс на работата, която всеки прави. Причината за това е, че може процесора x да смята взаимодействията за тела, които са много близо до много други тела и това са много взаимодействия. Процесорът у обаче съдържа основно тела, които са далеч от всички други и взаимодействията с тях при BH ще доведат до много по-малко работа поради апроксимации, което води до небалансирана работа между процесорите. За целта въвеждаме ORB.

3.1.2 Ортогонално рекурсивно балансиране ORB



Фигура 7: Примерен ORB с 8 процесора

Идеята на ORB е, че рекурсивно делим света на 2 части по по-дългата координата(разбиването на по-дългата координата намаля разхода на памет според [3]). Идеята е, че от двете страни на координатата сумата от теглата на телата трябва да са равни. Тогава на теория би трябвало да имаме равна работа на процесорите. Какво обаче ще използваме за тегло? За тегло взимаме броя взаимодействия между телата на последната итерация, а за първата итерация просто взимаме теглата да са единици, защото нямаме информация да твърдим, че не са равни. Тук разчитаме на това, че ако през последната итерация надали си се преместил достатъчно, така че броят взаимодействия да е много различен и вероятно и през тази итерация броят взаимодействия би бил приблизително еднакъв. Така, ако сме сметнали на едно тяло взаимодействията с 300 други тела сред, които може да има и апроксимации, то теглото му е 300. Използвайки теглата, можем да си разделим света на клетки равни на броя процесори, които ще съдържат по равно количество работа/взаимодействия. Ето един пример с 8 процесора:

На картинката се вижда, че на пръв поглед може процесори да имат неравен брой тела и това да значи повече работа, но реално телата по-близо до центъра могат да имат повече взаимодействия и затова така да са разделени клетките. Понеже на всяка итерация делим света на две равни по тегла части при последното делене всички процесори имат приблизително равна работа, т.е. сумата от теглата на телата, за които те смятат да са равни. Това разделяне може да се реализира с двоично дърво сравнително лесно. Всеки възел съдържа координатите на пространството, за което отговаря. Започвам в корена с множеството от всички процеси и търся, къде да разделя. Избирам координата и всяко тяло смята колко работа има от ляво и колко от дясно на координатата. После се синхронизират и сумират телата и всеки процес гледа дали разликата между работата отляво и цялата работа е ½. Ако е приблизително толкова се взима това за сцепване създава се нов възел на дървото, който има половината процеси и друг, който е с другата половина. Това се повтаря докато не останем с един процес, в който случай знаем, че този сектор от вселената е за този процес и той ще пресмята взаимодействията за телата в него. Този тип разделяне поражда архитектурата на хиперкуб, в която всеки възел е с $\log_2$ брой процеси на брой съседа. Аз това ще използвам за комуникация. Ще работя по следния начин. Нека всяко разделяне да има пореден номер. Така първото ще е едно. Второто разделяне отляво и отдясно ще са с 2 и т.н. Този пореден номер може и да се разгледа, като разстоянието от възела, който ще разделим до корена на ORB дървото. Той се използва, да можем да разделим процесите на ляво и дясно множество. Ние ще разглеждаме разделянията като комуникационни канали, а номерът им ще ни помага да намерим кой ни е съседа. Например при първото сцепване това, което ще направим е, че ще вземем всички процеси с нула в най-десния бит и ще ги сложим отляво на

разделянето, а процесите с единица вдясно. Така ще получим съответно при 4 процеса 00, 10 отляво и 11 и 01 отдясно. За целта се прави xor между единица и бита на поредния бит, който съответства на разделянето. Сега нека считаме за съсед всеки процес, който в двоичния си вид се различава само с един бит от нас. Тогава всеки процес има съсед от другата страна на всяко разделяне. В едно примерно ORB разпределение това ще са съседите на нулевия процес:

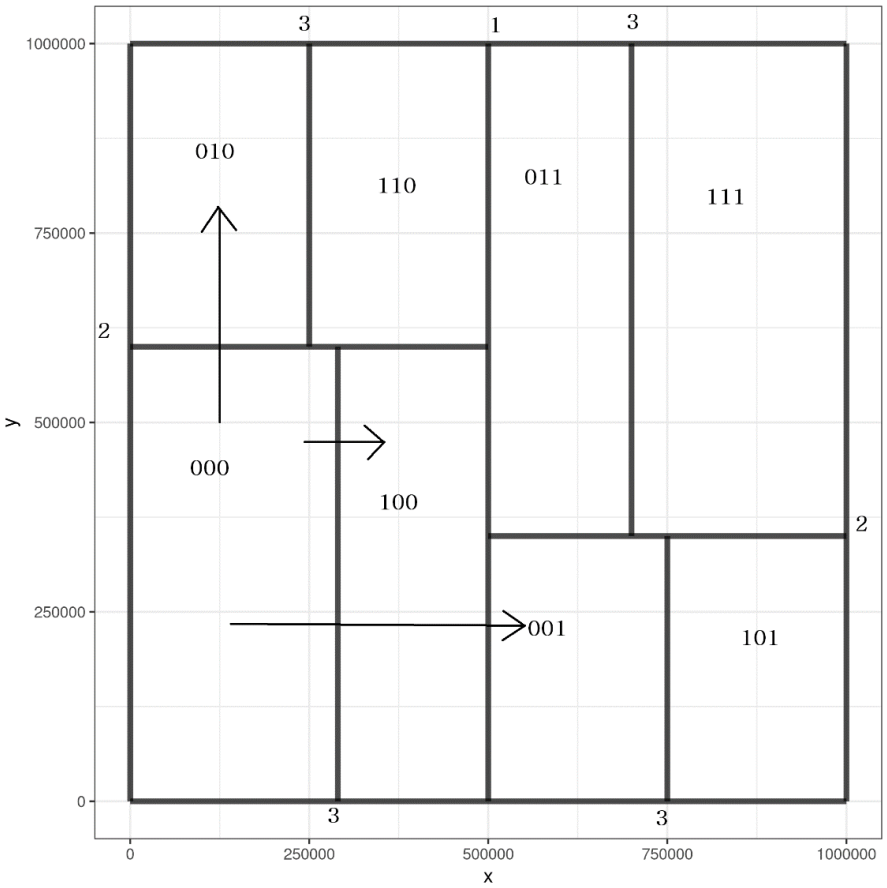
На картинката се вижда, че от всяко разделяне, в което 000 е участвал той си има съсед, с който може да си комуникира. Това важи и за всички други процеси и този факт ще използвам в основната част на алгоритъма при обмена на тела.

3.2 Построяване на Barnes-Hut дърво

3.2.1 Построяване на локално дърво

Като първа стъпка всеки процес взима своите тела, които е получил по време на ORB, и построява локално дърво по последователния алгоритъм описан в 2.1.2. Това дърво в момента ще съдържа само телата, за които нашият процес знае.

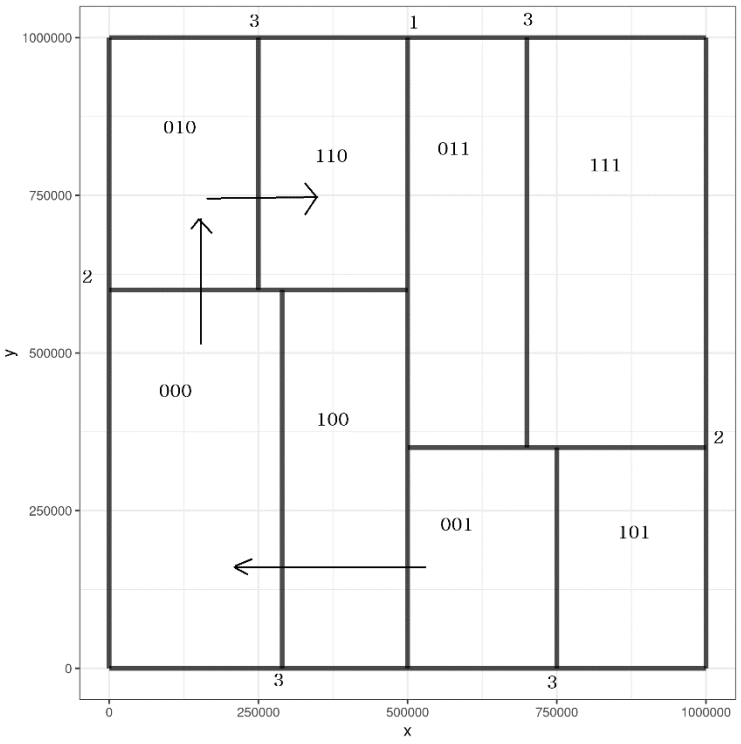
3.2.2 Обмен на телата



Фигура 8: Съседите на процес 0 при 8 процеса

Обмена на телата работи по следната схема. Всеки процес обхожда разделенията, в които е участвал по време на ORB-а. Това при 000 във фигура 7 са съответно тези линии през, които минават линии. Нали идеята е, че първо 000 е участвал в първото разделение по средата. После хоризонталното в лявата половина и накрая вертикалното до 000. Така процесите би трябвало в такъв ред да ги обходят. След това те обхождат своите локални BH дървета и добавят в една

опашка телата, които считат, че ще са нужни за елементите от другата страна на разделянето. Ако апроксимацията е достатъчна също можем да пратим нея. Така цялостно си намаляваме количеството комуникация между процесите. Проверката дали едно тяло е достатъчно далеч, да апроксимираме се извършва по следната формула: $dist * \theta > width$. Съответно θ е параметър за точност. Колкото по-малко число е толкова по-малко апроксимации правим и толкова ще са ни по-точни изчисленията. $dist$ може да се вземе по няколко начина. Това, което аз използвам е разстоянието от центъра на масата на множеството от телата, които мисля да апроксимирам до разделянето. Избрал съм това, защото така се избягват детониране на галактики според [3]. $width$ е ширината на клетката съответстваща на възела от BH дървото. Може да погледнете фигура 3, за референция. Ако уравнението е в сила добавяме в опашката едно фиктивно тяло, което нямаме с центърът на масата на множеството от тела, като координати и маса сумата от масите на тези тела. Като обходим локалното дърво и си подберем елементите за пращане си намираме съседа от другата страна на разделянето и пращаме на него тези тела и той си ги добавя в своето локално дърво. После ние от него получаваме неговите тела, които той е счел за важни, и си ги добавяме в нашето локално дърво. За да се избегне deadlock и да забие програмата съм го реализирал така че процесите от ляво на разделянето първо пращат и после приемат, а тези от дясно първо приемат, а после пращат. Въпрос обаче е дали всеки процес получава нужните му тела. Отговорът е да. Причината е, че на първа итерация моят съсед ми е дал жизненоважните тела за всеки процес от тази страна на разделянето. Аз после мога да преценя и да ги пратя на другите си съседи при обхождането на другите разделяния, а той да ги прати на свой съсед и т.н. Ето тук например ще използвам горната диаграма да покажа как 001 си комуникира с 110:



Фигура 9: Примерна комуникация

На фигурата се вижда как 001 говори първо със своя съсед 000, да му прати жизненоважните тела за процесите от другата страна на разделение 1. После при следващата итерация 000 праща тези тела на 010, ако прецени, че имат нужда от тях телата от другата страна на неговото второ разделяне. И накрая 010 праща каквото сметне за нужно на 110. Така може между всеки два процеса да се начертае път, по който телата могат да минат. Това обаче работи, само ако броят процеси е степен на двойката. Иначе примерно, ако бяхме със 7 процеса, то 101 нямаше да има съсед, защото 111 нямаше да го има, като число и той няма как да прати информация до 011 например. Затова всички тестова по кода са направени само с 2^n процеса.

3.2.3 Изчисления

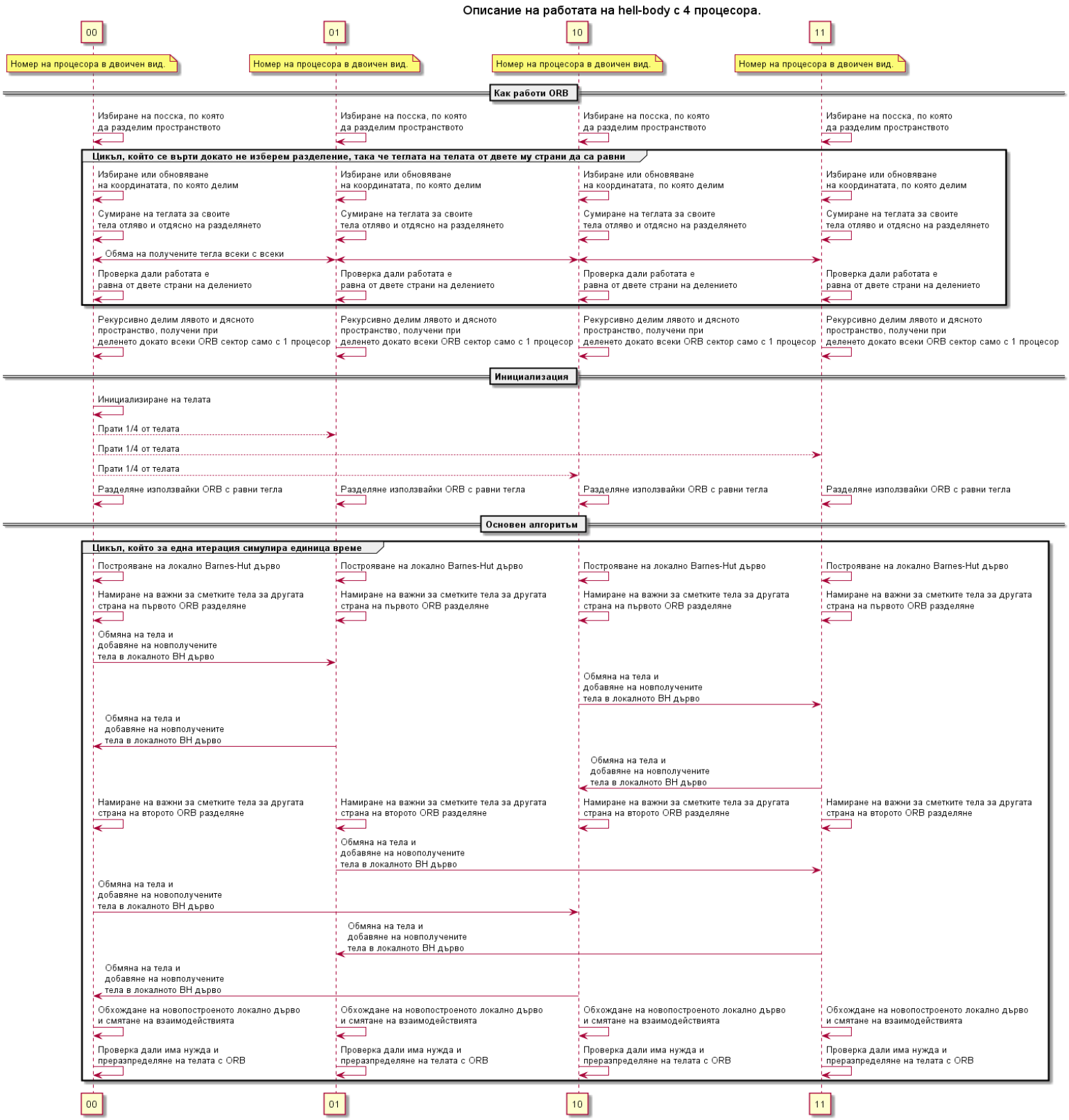
В тази фаза общо взето стават изчисленията. Вече всяко тяло би трябвало да е добавило в своето дърво жизненоважните му тела. Като следваща стъпка всеки процесор обхожда своите тела и изчислява взаимодействията с другите тела като обхожда дървото си. При всяко взаимодействие един брояч трябва да се вдига, да се отчете теглото за следващото преизчисляване на ORB. Като се сметнат силите после се обновява скоростта и накрая се обновяват позициите, като аз правя проверка дали някое тяло е излязло от света, ако е сменям скоростта му е я правя отрицателна и го връщам в света и изглежда все едно тялото се отблъсква от стените като билярдна топка.

3.2.4 Синхронизиране на телата и преизчисляване на ORB

В тази фаза телата се синхронизират наново между всички процесори вече с обновените координати и тегла, така че да може спокойно отново да се използват за преразпределянето им. След синхронизацията правя проверка дали има голяма разлика(повече от 5%, както в [4]) между броя взаимодействия на процесорите. Ако да, тогава ORB дървото се преизчислява от нулата. В противен случай използвам същото разпределение. В този случай, считам, че телата не са се преместили достатъчно, да има смисъл да изчислявам ORB дървото наново, което пести малко време.

3.3 UML диаграма на проекта

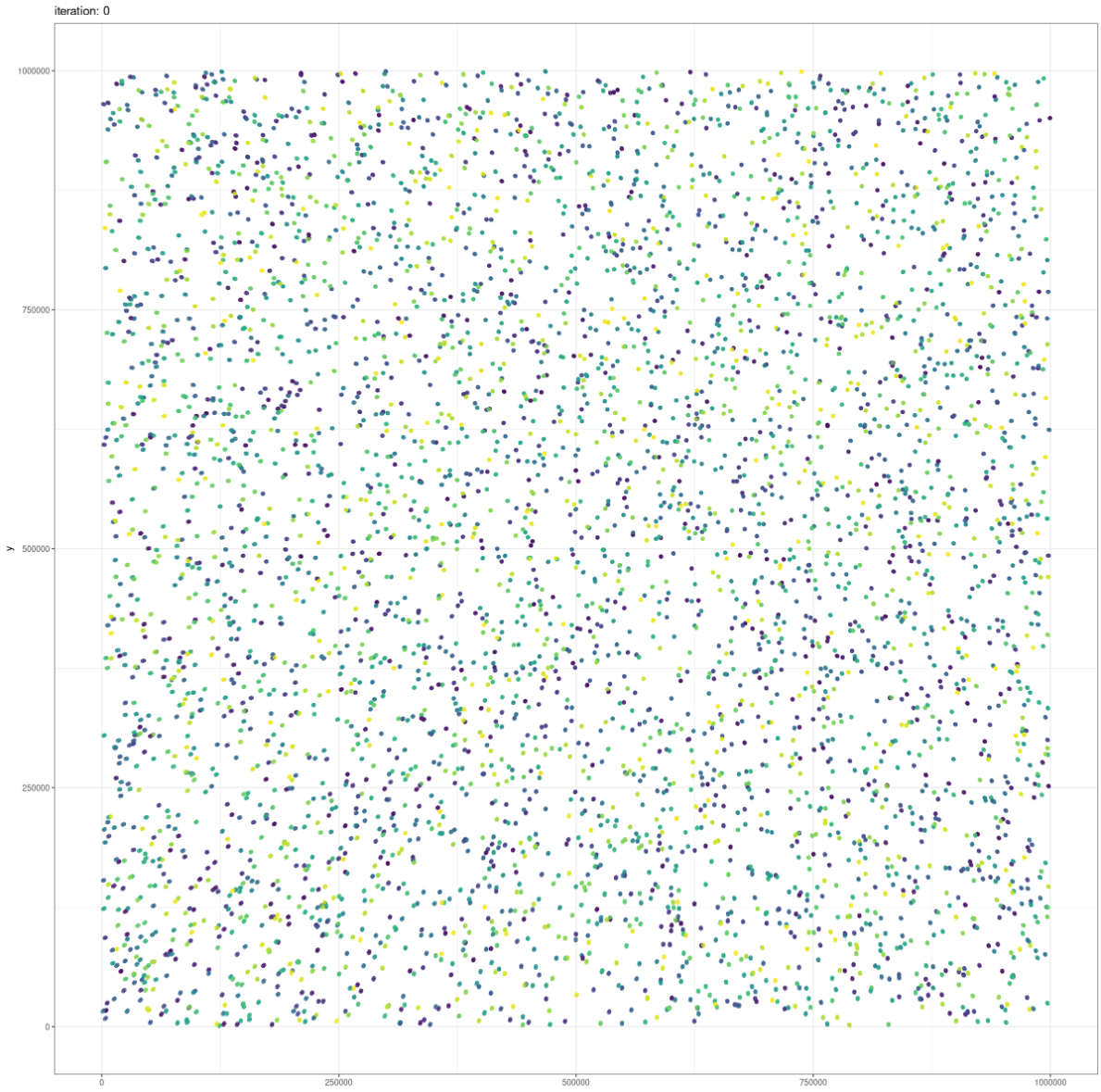
На следващата страница, фигура 10, съм сложил една UML диаграма, като съм се фокусирал основно върху паралелните части от алгоритъма. Неща като как точно работи последователния Barnes-Hut не съм описал, но идеята беше да се види основната идея на алгоритъма. Също така отгоре в секцията „Как работи ORB” съм сложил диаграма за ORB. Всеки път, като ORB е реферирян тази част от диаграмата се изпълнява. Самата програма реално започва в секцията инициализация и после продължава към секцията основен алгоритъм. Направих диаграмата с четири процесора с надеждата по-добре да се види реда на комуникация на процесорите по-явно. Самите комуникации с цел видимост са показани все едно са последователно, но реално са паралелни и работят по методите описани по-горе.



Фигура 10: UML диаграма на алгоритъма

3.4 Анимация

На фигура 11 съм сложил анимация на n-body симулацията си. Тя е изготвена, като съм използвал езика R и библиотеките gganimate и ggplot. Също така в нея участват само 5000 тела, защото картината става доста хаотична, а също машината, на която генерирах анимацията нямаше достатъчно RAM памет, да я генерира успешно с повече. Пробвах да генерирам анимацията на машината, на която пусках тестовете за паралелизъм, но пак имах проблеми и R не успяваше да генерира анимация с повече тела. Както и да е, исках само да спомена, че самите тестове на алгоритъма за ускорение са направени с доста повече тела.



Фигура 11: Анимация на n-body симулация с 5000 тела

3.5 Инструкции за употреба

3.5.1 Инсталиране на openMPI

Преди да започнете трябва да си инсталирате openMPI библиотеката и някой компилатор на c++, като например clang или gcc. Възможно е вашата машина вече да има инсталирани тези пакети. Ако ги няма, вероятно може да я инсталирате с вашия пакетен мениджър, ако сте под линукс, а за Windows има версия за cygwin според сайта на openMPI. За повече информация просто посетете сайта им.

3.5.2 Пускане на приложението

За стартиране на приложението трябва да изпълните две команди. Първо да компилирате програмата с командата:

mpic++ nbody.cpp

Може също така за по-добра работа да подадете флага -O3 на mpic++, което казва на компилатора да оптимизира кода. После полученият файл го пускате с командата:

mpirun -n <брой процеси> <името на получения при компилацията файл>

Това пуска програмата с n на брой процеса и записва изменението на координатите в coordinates.csv. Тези координати може да подадете на R скрипта в директорията animation, за да генерирате анимация. Този процес не съм го оптимизирал за лесна употреба. Ако искате анимация, ще трябва малко да промените R скрипта и да дадете имена на колоните в coordinates.csv файла. Това трябва да се оптимизира за по-нататък, допускайки, че някой би използвал програмата ми.

3.5.3 Параметри на програмата

Параметрите на програмата са под формата на глобални променливи в nbody.cpp файла. Като го отворите може да ги промените и модифицирате. Там са описани техните предназначение, но за пълнота ще ги изредя и тук:

- spaceX – максималната x координата
- spaceY – максималната y координата
- bodyCount – брой тела за симулацията
- MAX_START_VELOCITY – максимална начална скорост на телата
- MIN_START_VELOCITY – минимална начална скорост на телата.
- MASTER – това е променлива, която обозначава процес 0. Тя се използва само в началото за първоначалното разпределяне на телата. Нататък вече не се ползва и няма смисъл да я променяте.
- TIME – количество време, което ще симулираме или по-точно брой повторения на алгоритъма.
- DELTAT – определя колко точно е една мерна единица в нашата симулация, т.е. колко време е изминало при една итерация на алгоритъма

- THETA – параметър определящ точността на Barnes-Hut алгоритъма и по-точно колко често ще правим апроксимации. По-ниска стойност значи, че е по-точен алгоритъма. По-високи обаче значи, че програмата работи по-бързо.
- MAX_MASS – максимална начална маса на телата
- MIN_MASS – минимална начална маса на телата
- BODIES_PER_LEAF – колко тела се намират в листата на Barnes-Hut дървото. При достатъчно малко тела е по-бързо директно да сметнем взаимодействията вместо да обхождаме дървото постоянно. Прави и самото BH дърво по ниско.
- ORB_SPLIT_ERROR – променлива, която трябва да е между 0 и 1 и която определя колко трябва да е близко до ½ работата от двете страни на разделянето при ORB. При по-ниски стойности стойности е по-равномерно разпределена работата между процесорите.
- ORB_REBUILD_WEIGHT – стойност определяща колко често да строим ORB дървото. Тя трябва да е между 0 и 100 и определя колко процента е разликата между работата на два процесора, за да считаме, че е нужно да се построи дървото наново.

4. Анализ

4.1 Характеристики на тестовата машина

Машината има два процесора Intel®Xeon® CPU E5-2660 0 @ 2.20GHZ. Всеки процесор има по 8 ядра, което прави общо 16 ядра на машината и всяко ядро поддържа по 2 нишки, което означава, че е има смисъл да си пускам програмата с до 32-а процеса.

От към памет машината има 64 gb RAM, по 32K за L1d и L1i кешовете. 256K за L2 кеш и 20480K за L3 кеш.

4.2 Постигнати резултати

За изчисляването на моите резултати съм засичал цялостното време за работа на програмата от пускането на първият процес до приключването на работата на последния. Правил съм по 3 опита и съм взимал минималното време за работа то трите. При моя алгоритъм грануларността се задава основно от броя тела, защото тя е равна на броя тела делено на броя процеси: N/N_{proc}. Аз създадох таблица с моите резултати, в която съм добавил 4 основни параметъра:

- p – брой процеси
- n – брой тела
- t – брой итерации
- θ – понеже е важно за скоростта на изчисленията добавих и BH параметъра, който използвах

#	*p*	*n*	θ	*t*	$T_1^{(1)}$	$T_1^{(2)}$	$T_1^{(3)}$	T_1=min()	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	T_p = min()	$S_p = T_1/T_p$	$E_p = S_p/p$
1	2	100000	0.2	10	***479.5***	***477***	***473***	**473.29156**	*263*	*261.2*	*255.2*	255.216204	1.854473002	0.927236501
2	4								*137.8*	*133.3*	*140.6*	133.338106	3.549559644	0.887389911
3	8								*74.97*	*74.37*	*73.62*	73.620645	6.428788555	0.803598569
4	16								*43.68*	*41.43*	*42.62*	41.427862	11.42447467	0.714029667

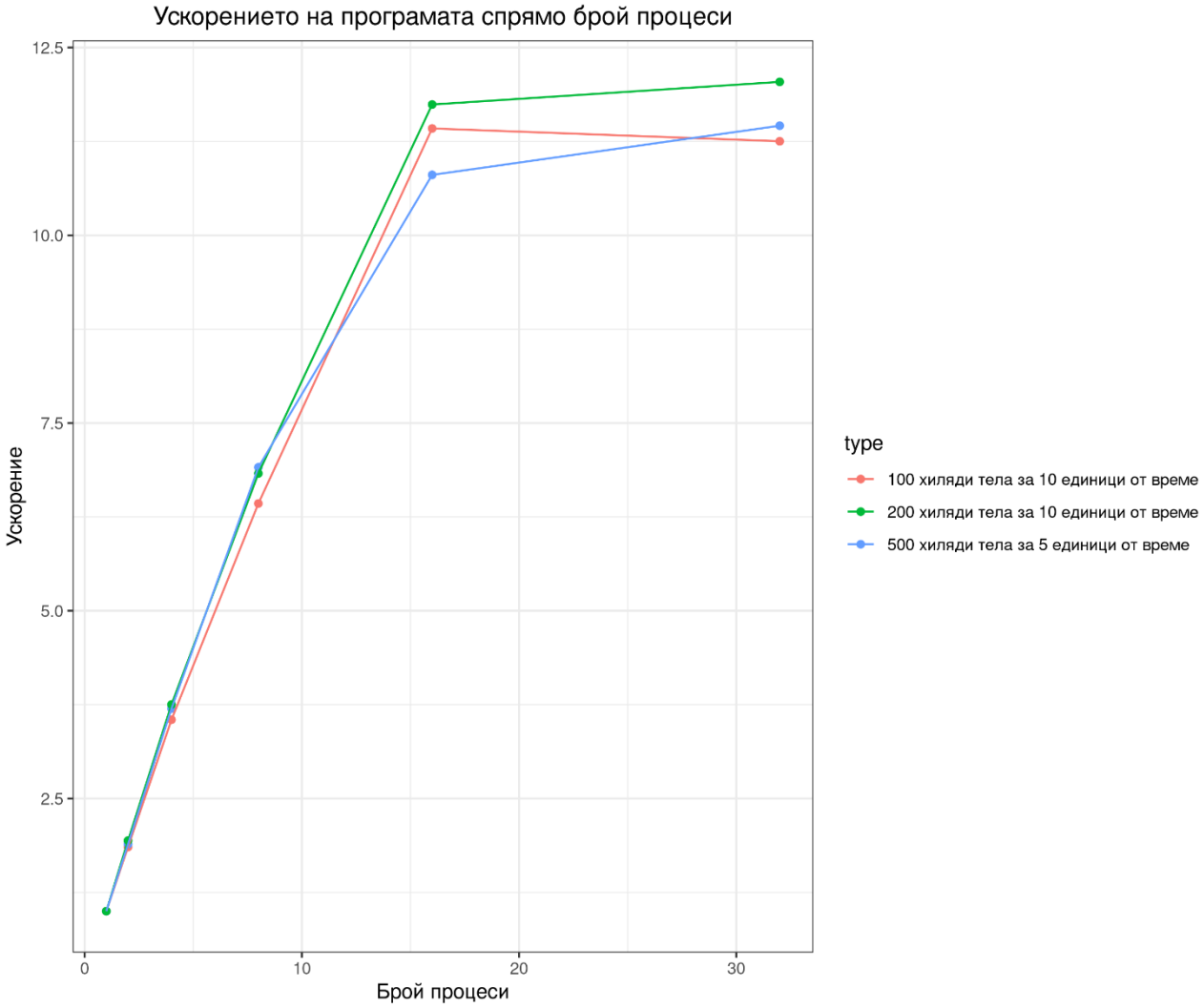
5	32								*43.57*	*42.79*	*42.06*	42.058651	11.2531322	0.351660381
6	2	200000	0.2	10	***963.5***	***959***	***958***	**958.25322**	*505.6*	*493.8*	*504*	493.819269	1.94049378	0.97024689
7	4								*260.4*	*255.4*	*256.5*	255.397725	3.752003742	0.938000936
8	8								*141.4*	*140.3*	*142.3*	140.318899	6.829110169	0.853638771
9	16								*81.61*	*89.43*	*89.91*	81.610529	11.7417842	0.733861513
10	32								*80.36*	*79.57*	*79.94*	79.567834	12.04322365	0.376350739
11	2	500000	0.2	5	***893.3***	***902***	***889***	**889.01017**	*469.8*	*473.7*	*469.8*	469.804249	1.892299116	0.946149558
12	4								*240.4*	*243.7*	*240.8*	240.415161	3.697812406	0.924453102
13	8								*129.4*	*128.6*	*129*	128.642401	6.910708741	0.863838593
14	16								*82.52*	*84.44*	*82.27*	82.266984	10.80640278	0.675400174
15	32								*77.57*	*82*	*79.55*	77.57399	11.46015778	0.358129931

Фигура 12: Резултати на програмата

4.3 Анализ на таблицата

Като цялото резултатите от програмата не са лоши според мен, но не са оптимални. Има какво да се желае от гледна точка на ускорението. При 16 процеса това води в общия случай почти до 12 пъти ускорение. Цялостно мисля, че това разминаване може да се дължи на това, че използвам обмен на съобщения в рамките на една машина. По принцип това е разпределена програма и вероятно би било по-оптимално да работя както са работили в [2], [3] и [4] на системи от компютри или по-точно клъстери, да постигна по-добри резултати. Също така както видяхме от [4] има доста методи да се оптимизира задачата и да се намали работата извършена при пресмятането на ORB и на построяването на BH дърветата, като се задели кеш за част от данните или като ORB-а обновява само една част от дървото си, а не цялото. Всичко това може да са оптимизации, които си струва да се имплементират, да стигна до по-добри ускорения. Самата имплементация в на Salmon в [3] е доста сходна с моята и при него от на фигура 5 се вижда, че ускорение има до 512 процесора. Разбира се може броя тела да не е показателен, защото неговите изследвания са проведени на доста по-стара машина, за която вероятно 20000 тела са били доста по-голямо количество сравнено с модерните процесори. При моите резултати се вижда ясно и че ефективността доста бързо спада, като при 16 процесора вече е на 60%-70%. Друга причина за това мисля, че може да е количеството тела, с които съм тествал. За да проверя

това реших да пусна на сървъра тестове за 16 и 32 процесора и 8.8 милиона тела. Резултатите бяха, че има разлика от близо една минута. Отне му на в първия случай 6:35, а във втория 5:40.

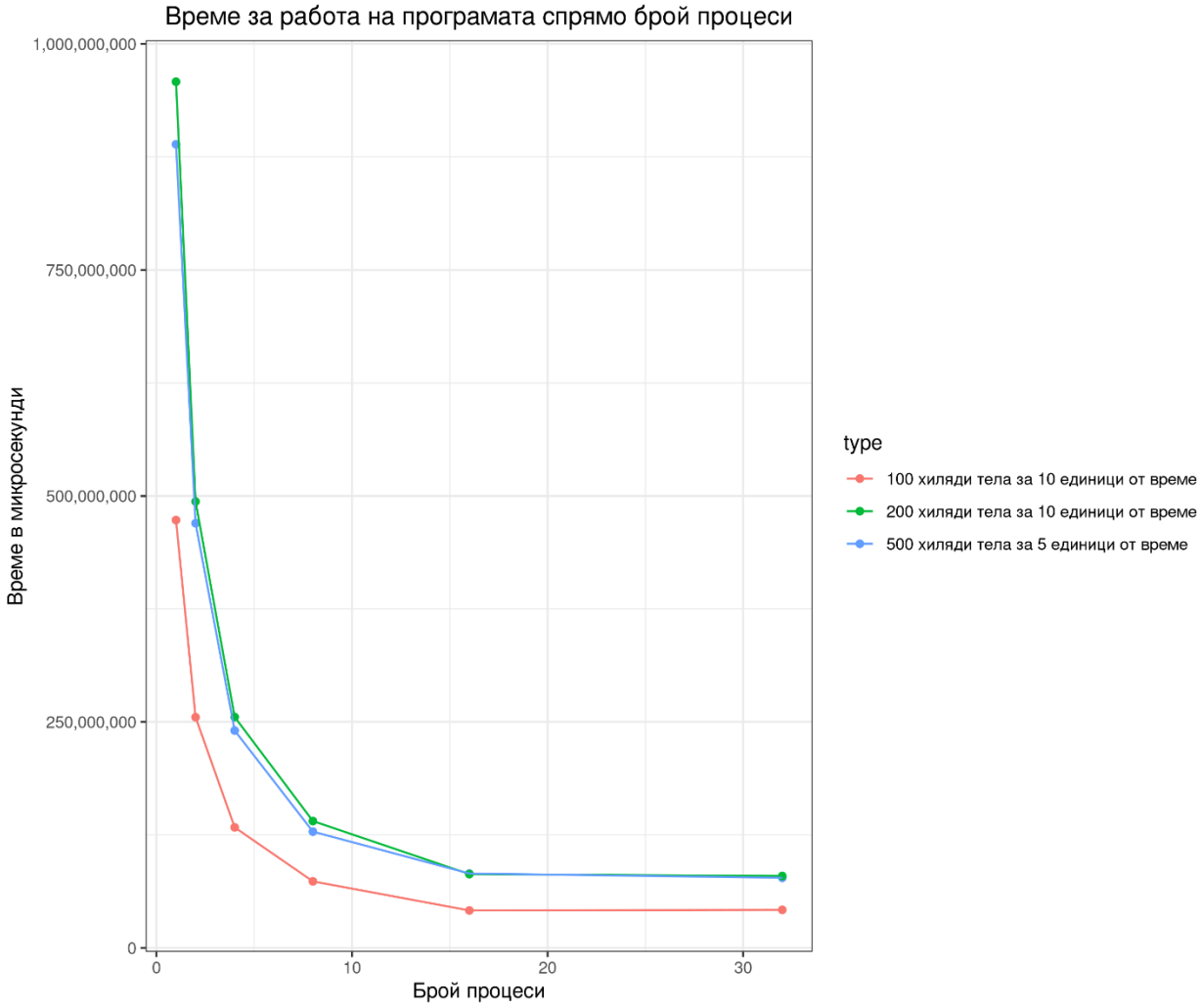


Тази разлика беше пропорционална на разликата видяна по-горе, но определено се вижда по-значително разминаване във времето отколкото на моите тестове. Причина да не пусна тестове и за 1 процесор на 8.8 милиона тела е, че нямам нервите да го чакам, а и не знам колко време това би товарило тестовата машина, на която работят и други хора, но все пак има някакъв потенциал да получа по-добро ускорени, макар че не знам с колко обаче. Също така в тези тестове, ако са само по една итерация щеше да е трудно да има ефект ORB алгоритъмът върху разпределението на работа, защото на първата итерация той общо взето не прави нищо. Иначе ето още няколко диаграми, които генерирах за по-нагледна репрезентация на данните от таблицата:

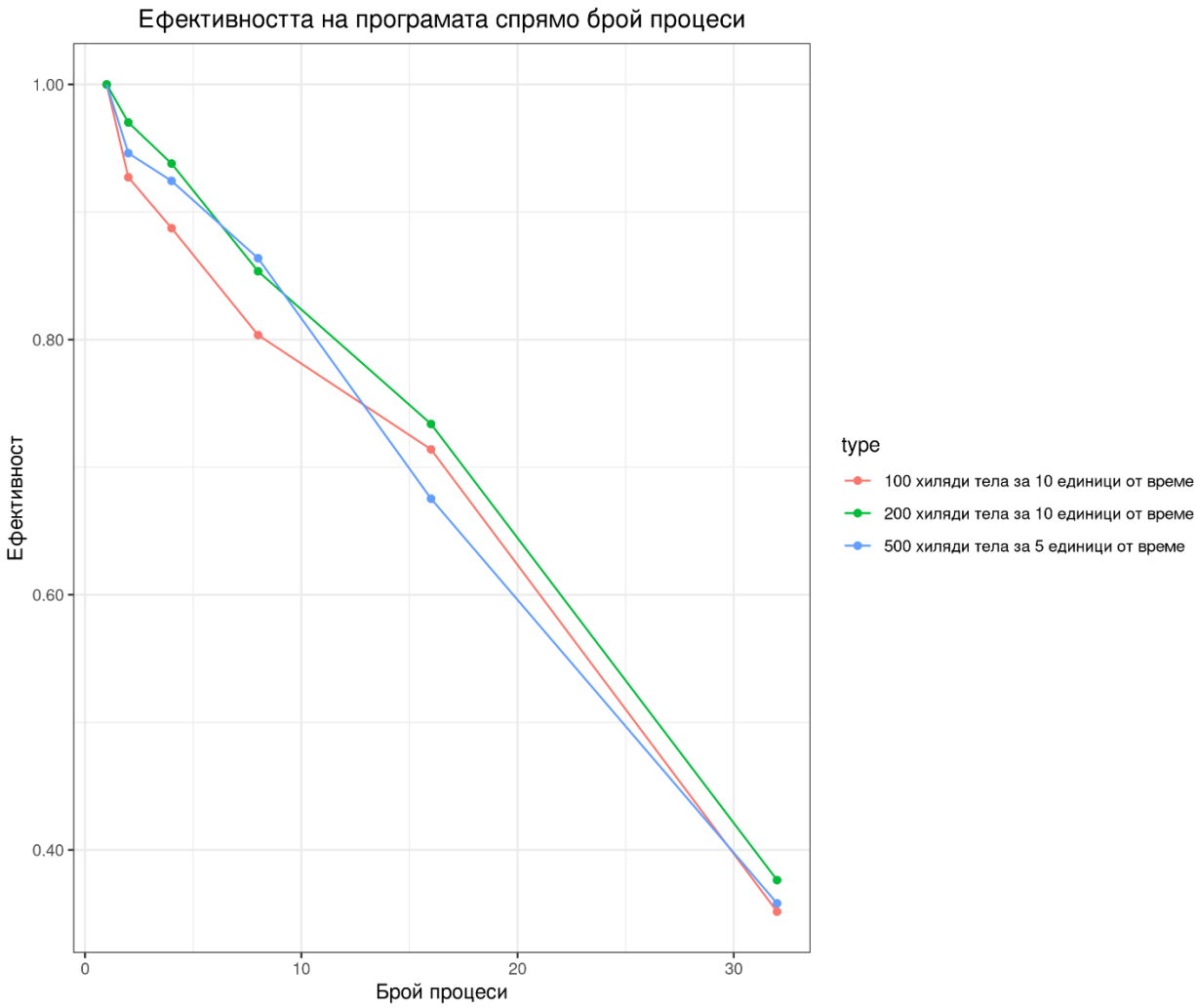
Фигура 13: Диаграма на времето на работа на процесите

На тази диаграма доста добре се вижда, че разлика между работата при 32 процеса и тази за 16 почти няма. Това както казах може да е и до количеството тела. Друга хипотеза, която имах е, че не съм напълно сигурен как хипернишковата технология влияе в случая на обмен на съобщения. Ако бяха отделни нишки, със сигурност щеше да влезе в сила, но сега като са отделни процеси, не знам дали не е от значение. Опитах се да задълбая в темата повече, но в информацията, която намерих беше оскъдна и крайно недостатъчна, да ми потвърди дали хипотезата ми е вярна.

В следните фигури можем да видим и ускорението, което съм получил по-нагледно, както и ефективността:



Фигура 14: Диаграма на ускорението

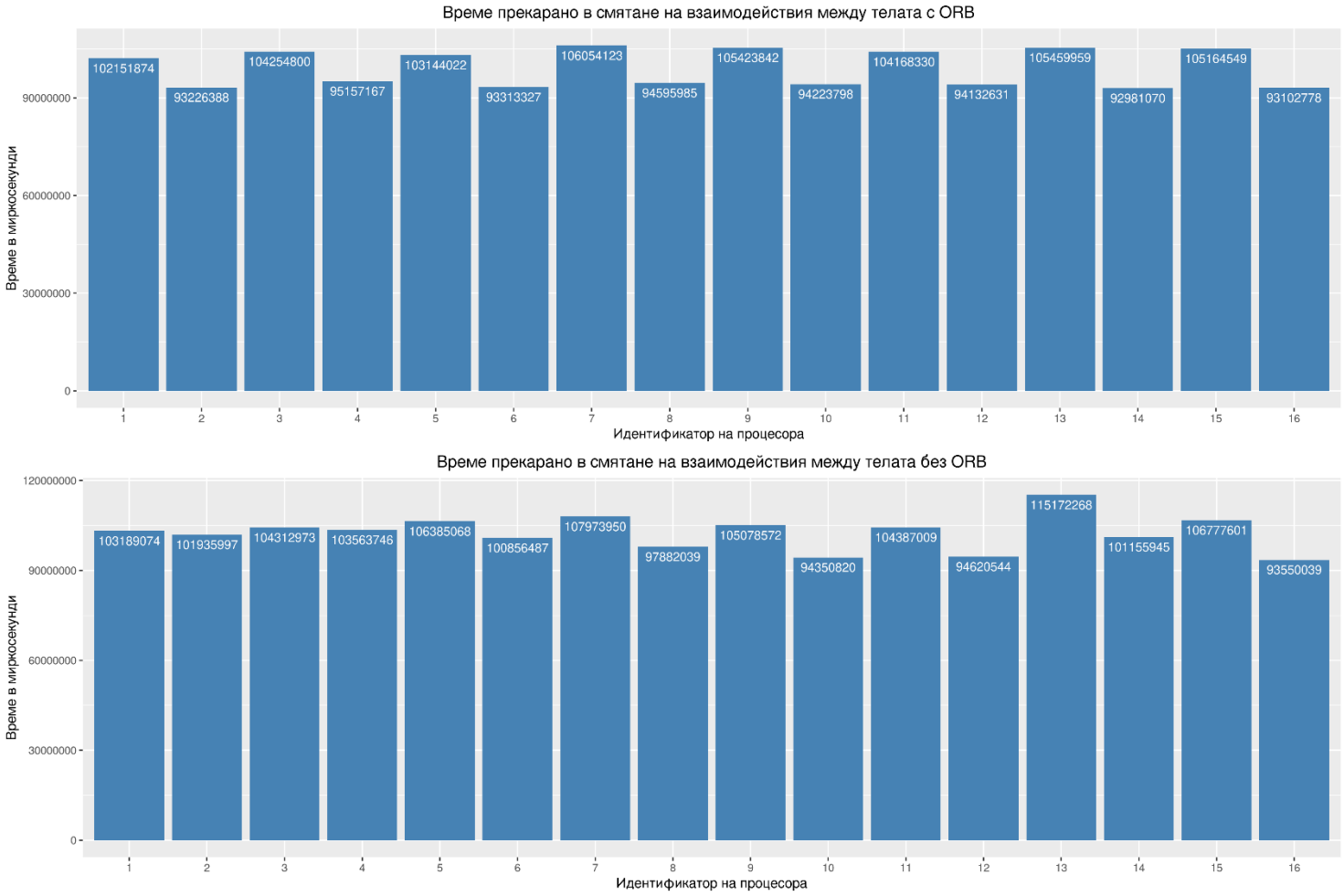


Фигура 15: Диаграма на ефективността

Горните две диаграми пак наблюдават сравнително плавно увеличаване на ускорението и плавно намаляване на ефективността докато не стигнем 32 процеса, където рязко спада ефективността и ускорението нараства с малко. Резкия спад вероятно се дължи и на това, че не мога да тествам с брой процесори, които не са степен на двойката, защото тогава комуникацията използваща модела на хиперкуб не би работила и така не виждам реално, кога ускорението спира да се покачва толкова добре.

4.4 Анализ на разпределението(ORB)

Тук реших да погледна колко работа върши всеки един от моите процесори при едно пускане на програмата и така да преценя колко е ефективно разпределението и дали има полза от него поне при моята имплементация. За целта пуснах програмата си с ORB и без за 50 итерации с 16 процеосора. Като без ORB пак работеше ORB алгоритъма, защото не мога да разделя телата без него и той е жизненоважна част от кода, но просто сложих теглата да се задават, като единици след, като се преизчислят, което би трябвало да разпредели приблизително по равен брой тела на процес. Понеже ORB-а реално пак работи и в двата случая реших да се фокусирам само върху времето нужно за изчисленията на взаимодействията на телата, като така ще се абстрахираме от всякакъв overhead и ще видим дали в един от двата случая работата е по-добре разпределена.

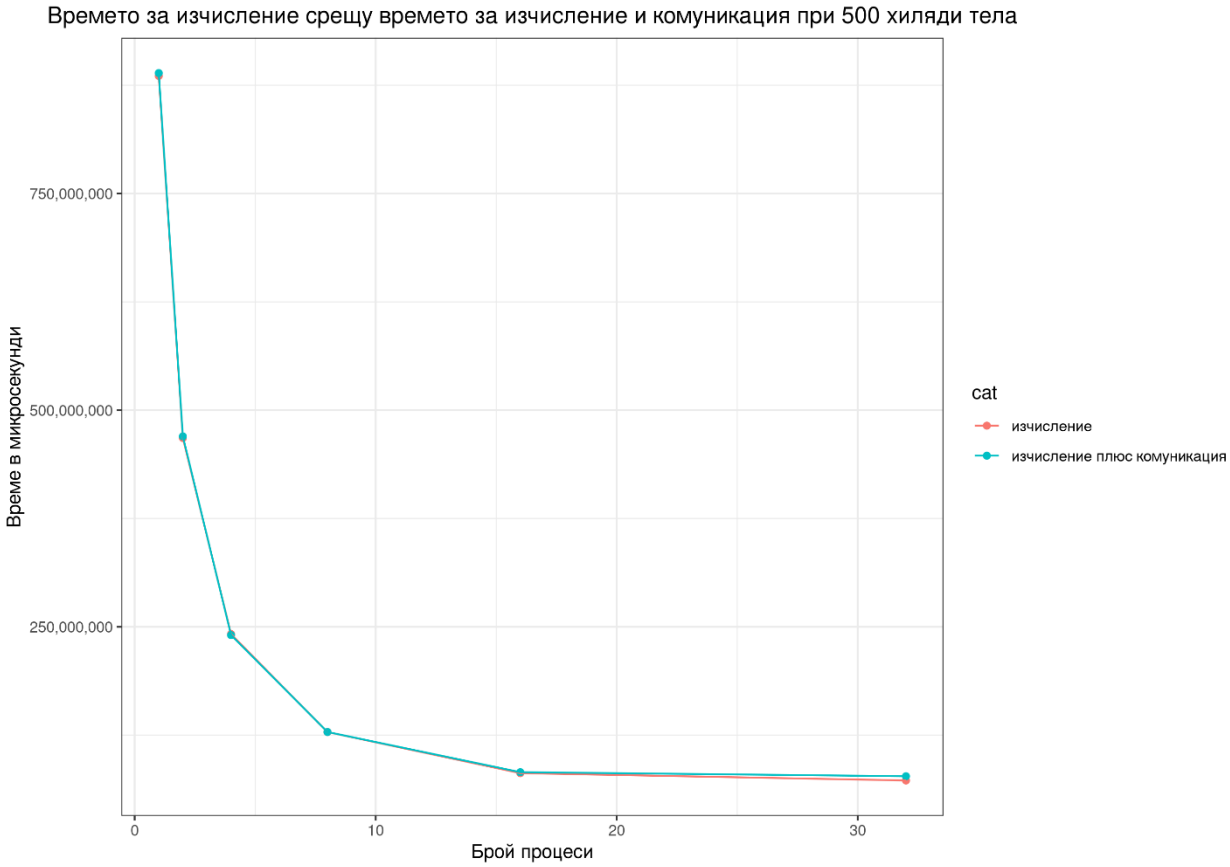


От диаграмата първо се вижда, че ORB разпределя работата между процесите равномерно. Другото, което е съществено е, че разлика в работата с и без ORB има. Процесор 13 без ORB е доста по-натоварен спрямо останалите. Разликата в ситуацията може и да не е съществена, но такава има. При по-голямо количество тела, по-голям свят и повече процеси вероятно бихме имали доста по-съществена полза от ORB-а, но дори тук се вижда полза, макар и малка.

4.5 Анализ на времето за комуникация

Реших да направя диаграма, която сравнява времето за работа на програмата, като цяло с времето за изчисленията на взаимодействията:

Фигура 16: Разлика в работата със или без ORB



Тук се вижда, че времето за обмен на телата плюс времето за изчисления при 500 хиляди тела е почти еднакво. Знаем и че телата са равномерно разпределени от 4.4. Тогава по пътя на логиката можем да заключим, че вероятно или телата са твърде малко за 32 процеса и това води до липсата на ускорение. Също може да е проблем и фактът, че имплементираният алгоритъм е за разпределени системи с обмен на съобщения и това може да води до проблеми. Поне аз не виждам, защо времето е почти същото с 32 и 16 процеса, имайки предвид горните съображения.

5. Заключение

5.1 Мои мисли върху проекта

Цялостно този алгоритъм беше доста предизвикателен за реализация. Беше много интересно да го реализирам и тествам. Голям проблем се оказа библиотеката, която си избрах. Всякакви грешки трябваше с четене да си ги откривам. Вече имам чувството, че наизуст знам кода на тази програма след толкова прочити. Вероятно, ако можех да върна времето назад бих го писал на някой друг език или поне бих пробвал да направя реализация с нишки и споделена памет, което би било и по-оптимално за хардуера. Сега имаше много загубено време в поправяне на грешки. Доста нерви щеше да ми спести това да имам един дебъгер. Реално има няколко, които библиотеката поддържа, ама всички са платени освен gdb, с което пробвах, ама не успях да се оправя със настройването. Както и да е, цялостно съм доволен от резултатите и анимацията изглежда много готино. Нямам най-великото ускорение, на което бих се надявал и горе описах на какво мисля, че това може да се дължи, ама все пак не мисля, че е лошо поне на пръв поглед. Също така

разпределението работи, както видяхме и разпределя работата равномерно и времето за комуникация и строене на дърветата е малко, така че се вярвам, че според резултатите е малко вероятно грешката да е в кода, а по-скоро да засяга хардуера, на който го пускам.

5.2 Подобрения за в бъдеще

Цялостно мога да пробвам да реализирам проекта със споделена памет и да сравня как се сравнява, като ускорение. Също така бих могъл да имплементирам оптимизациите в [4], т.е. да пазя част от данните в някакъв кеш за по-бърз достъп и да права инкрементални промени по BH дървото вместо да го строя наново при всяка итерация. Също така от кода ми от гледна точка четимост има какво да се желае. В момента всичко е един голям файл. Ще е добре да го разбия на няколко файла да си го преструктурирам и подробно да огледам какво точно ми трябва. Също така бих могъл да оптимизирам логовете да могат да се включват и изключват или да има различни нива за логове, като ниво на дебъг и някакви други по-стандартни за по принцип. Както споменах мога да оптимизирам малко и генерирането на анимации, да става по-лесно, ама това е за друг път. Засега е това от мен.

6. Източници

- [1] Damon Binder, Simulating Gravity, (http://djbinder.com/articles/NBody/)
- [2] Guy Blelloch и Girija Narlikar, A Practical Comparison of N-Body Algorithms, American Mathematical Society, 1997, (https://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/dimacs-nbody.pdf)
- [3] John K. Salmon. Parallel Hierarchical n-body methods, 1991, (https://thesis.library.caltech.edu/6291/1/Salmon_jk_1991.pdf)
- [4] Pangfeng Liu и Sandeep N. Bhatt. Experiences with parallel N-body simulations. Част от шестия годишен ACM симпозиум по паралелни алгоритми и архитектура (SPAA’94), страници 122–131, 1994, (https://www.csie.ntu.edu.tw/~pangfeng/papers/IEEE%20TPDS%20Nbody%20experience.pdf)