МЕТОД СВЯЗАННЫХ ЯЧЕЕК С СОРТИРОВКОЙ ВЗАИМОДЕЙСТВИЙ ДЛЯ ВЕКТОРИЗОВАННЫХ РАСЧЕТОВ В МОЛЕКУЛЯРНОЙ ДИНАМИКЕ

Матвиенко С.А[1,2], Алемасов Н.А.[2], Фомин Э.С[2]

[1]Новосибирский государственный университет
[2]Институт цитологии и генетики СО РАН

matvienko90@gmail.com, alemasov@bionet.nsc.ru, fomin@bionet.nsc.ru

Аннотация. С точки зрения эффективности метод связанных ячеек (LC - linked cells) с сортировкой взаимодействий (IS - interaction sorting) является достойным конкурентом широко используемому улучшенному методу Верлет таблицы (IVT - improved Verlet Table). При применении метода LC + IS для плотных систем с высокой мобильностью атомов (например, жидкая фаза) и относительно большим радиусом взаимодействия атомов ($r_{cutoff} > 3r_{vdw}$) достигается заметно большая, до 15%, скорость выполнения молекулярной динамики (МД) относительно метода IVT. В данной работе предлагается дальнейшее улучшение метода LC + IS для векторизованных расчетов, которое использует тот факт, что для алгоритмов, написанных с использованием SSE-векторизации отсутствует необходимость точной сортировки атомов и вполне достаточной является сортировка с точностью до локализации атома в том или ином блоке атомов размерностью в 4 элемента. Использование блочной сортировки позволяет не только снизить затраты на сортировку, но и векторизовать сам код сортировки, что немаловажно для LC + IS, где сортировка вносит существенные накладные расходы. Как показано в данной работе подобное улучшение метода дает дополнительный прирост производительности от 9% до 45% относительно оригинального метода LC + IS. Приводятся результаты сравнения эффективности предложенного метода с другими современными методами расчетов, предназначенных для решения этой же задачи.

Ключевые слова

Молекулярная динамика, ближние несвязующие взаимодействия, метод связанных ячеек, метод Верлет таблицы, сортировка взаимодействий, SSE-векторизация.

Введение

Расчеты ближних несвязующих взаимодействий в молекулярной динамике (МД) занимают существенную, до 90%, часть от общего времени выполнения, что обусловлено следующими факторами:

- по сравнению с расчетами связующих взаимодействий резко (в десятки раз) возрастает число взаимодействий, приходящихся на один атом, что обусловлено необходимостью учета влияния всех атомов, находящихся в сфере радиуса отсечения взаимодействий r_{cutoff};
- заметно увеличивается неоднородность данных (в области взаимодействия оказываются атомы разных типов);
- в потоке данных, доставленных из памяти на вычислительное устройство, высока доля незначимых пар атомов, то есть таких, которые при проверке расстояний приходится отбрасывать;
- существенно возрастает количество промахов кэша, поскольку атомы, хранимые в соседних ячейках памяти компьютера, в силу своего непрерывного движения в процессе моделирования могут существенно разойтись по координатам в пространстве и выйти из области учета взаимодействий.

Таким образом, поток данных, попадающий на вычислительные устройства на этапе расчетов несвязующих взаимодействий, в случае неудачного подбора алгоритмов, отсутствия их адаптации для конкретных вычислительных платформ характеризуется низкой плотностью «полезных» данных, обрабатываемых в единицу времени, что неизбежно сказывается на эффективности расчетов.

Высокую эффективность расчетов в молекулярной динамике могут обеспечить только адаптивные схемы организации вычислительного процесса, которые позволяют существенно увеличить вероятность нахождения нужных данных в потоке за счет подстраивания схемы вычислений под текущее состояние моделируемой системы и платформу выполнения. В настоящее время такие адаптивные схемы организации вычислений используют метод связанных ячеек (LC - linked cells) [1] для поиска ближайших соседей, метод сортировки взаимодействий (IS - interaction sorting) [2] для ограничения доли незначимых пар атомов в соседних ячейках, метод Верлет таблицы (VT - Verlet table) [3] для хранения временно взаимодействующих пар, методы периодического переупорядочения атомов (LCR - linked cell based reordering) [4] для минимизации промахов кэша, автоматическую, «на лету», подстройку параметров расчетной схемы [5] под максимальную эффективность путем контролирования времени работы различных шагов алгоритма. Кроме того, следует упомянуть целый пласт идей, направленные на уменьшение накладных расходов, связанных с обработкой Верлет таблицы: использование перестройки таблицы только для данных потерявших актуальность (частичная перестройка [6, 7]), использование сортировки взаимодействий для уменьшения времени построения Верлет таблицы ([8]), уменьшение конфликтов потоков за данные в многопоточном выполнении (метод локальных Верлет таблиц [9]) и даже идею «воображаемой» Верлет таблицы, то есть такой таблицы, которая имеет только интерфейс, а нужные данные вычисляет по запросу без обращения в память (метод псевдо-Верлет таблицы [10]).

Следует заметить, что наиболее распространённая схема обработки данных в МД, основанная на том, чтобы сперва найти и собрать все взаимодействующие пары атомов в некоторый список/таблицу (Верлет таблица), а затем рассчитать между ними все взаимодействия, конфликтует с текущими тенденциями развития вычислительной техники. Конфликт возникает из-за обязательного присутствия промежуточного объекта (Верлет таблица) с неизбежной косвенностью обращения к данным через нее, с отсрочкой вычислений и потерями времени на пересылку (загрузка данных в/из памяти), с увеличением количества промахов кэша для таблицы, содержащей данные «про запас» для обеспечения валидности таблицы на большее, чем один, количество временных шагов моделирования, с увеличением конфликтов потоков выполнения за данные таблицы и невозможностью векторизации извлечения данных (косвенность оперирования с Верлет таблицей и разбросанность данных по памяти).

Недостатки метода Верлет таблиц хорошо демонстрируются в результатах нашей работы [11]. Сравнительное тестирование метода Верлет таблиц и метода связанных ячеек с сортировкой взаимодействий на процессоре Intel ManyCore [12] показало низкую эффективность метода Верлет таблиц при использовании большого (>16) числа вычислительных ядер. Таким образом, для того, чтобы полностью воспользоваться возможностями современной техники, требуется устранить промежуточный объект - Верлет таблицу - и использовать взамен «прямые» подходы, основанные на методе связанных ячеек.

Как упоминалось выше, метод LC + IS является достойным конкурентом для метода IVT, демонстрирующем большую $\approx 15\%$ эффективность при расчете плотных систем с высокой мобильностью атомов. Как показано в [13] метод превосходит IVT как в последовательном, так и в многопоточном выполнении, как без, так и с SSE-векторизацией кода. Тем не менее, в [13] отмечалось, что метод LC + IS не в полной мере использует возможности векторизации, что обусловлено невозможностью эффективной векторизации алгоритмов сортировок. Относительные потери на сортировку в последовательной версии не превышают 5-10% и повышаются до 15% для SSE версии. Метод сортировки взаимодействий, обладая доказанной эффективностью для стандартной архитектуры x86, может не дать эффекта при его использовании на векторных процессорах с большой шириной вектора, то есть при переходе от SSE (ширина вектора равна 4 элементам) к AVX (8 элементов) или даже к большим по ширине вектора, проектируемым в настоящее время системам команд AVX-512, AVX-1024 (16, 32 элемента). Наиболее существенной проблемой является необходимость многочисленных сортировок данных, и далеко не очевидно, что алгоритмы сортировок могут эффективно использовать потенциал векторизации с увеличенной шириной вектора. По крайней мере, на первый взгляд совершенно не ясно, каким образом могут быть векторизованы наиболее быстрые (с вычислительной сложностью $O(NlogN)$) и популярные в последовательных программах алгоритмы сортировок, такие как быстрая сортировка [14], сортировка слиянием [15] и прочие алгоритмы. Таким образом, увеличение ширины вектора в следующих поколениях процессоров неизбежно ставит вопрос о пригодности метода LC+IS в будущем для молекулярной динамики.

Данная работа показывает возможность ликвидации указанного недостатка метода LC+IS для векторных расчетов. Основная идея подхода заключается в том, что можно выполнять сортировку данных с точностью только до размещения атома в пределах того или иного блока данных (который по сво-

ему размеру должен совпадать с шириной вектора данных). Необходимость поэлементной сортировки, которую обеспечивают алгоритмы полной сортировки, такие как быстрая сортировка и прочие, отсутствует, поскольку все элементы вектора данных обрабатываются совместно. То есть, чем выше ширина вектора данных, тем меньшие требования налагаются на точность сортировки и тем более эффективным становится шаг упорядочения данных.

Анализ существующих методов сортировок показывает, что среди всех алгоритмов сортировок наиболее перспективным для векторизации в методе сортировки взаимодействий является блочная (или корзинная) сортировка. Алгоритм блочной сортировки имеет вычислительную сложность $O(N)$, хотя и требует знания природы сортируемых данных. Именно по последней причине данный алгоритм практически не используется в качестве обобщенного алгоритма в стандартных библиотеках, которые должны быть независимы от природы сортируемых данных.

Основные особенности алгоритма сортировки взаимодействий, которые необходимо учитывать, следующие:

- Сортировке подвергаются массивы с весьма небольшим числом элементов в них. Действительно, среднее число молекул воды на любую пространственную ячейку размером $10 \times 10 \times 10$ Å^3 при нормальной плотности не превышает ≈ 30 (полное число атомов ≈ 90).
- Общий объем необходимых сортировок значителен, то есть число небольших по 90 элементов последовательностей велико. Действительно, общее число определяется числом ячеек, на которые разбита пространственная область $[N_x, N_y, N_z]$ и числом пар, которые образует любая ячейка с соседями, то есть 26. Таким образом, полное число массивов для сортировки равно $26 \times N_x \times N_y \times N_z$, что для среднего расчета достигает величины в несколько десятков тысяч. При этом такие сортировки требуется выполнять на каждом шаге МД.
- Все атомы, попадающие в любую пространственную ячейку всегда ограничены по координатам пределами ячейки, что позволяет для каждой ячейки статически задавать условия размещения того или иного атома в ту или иную корзину.
- Все атомы в процессе своего движения не могут близко подходить друг у другу в силу действия потенциала отталкивания $O(r^{-6})$, и таким образом они распределяются в пространстве достаточно равномерно, что в свою очередь приводит к разномерности заполнения блоков.

Таким образом, для задачи сортировки атомов в рамках метода связанных ячеек анализ данных показывает, что метод блочной сортировки вполне пригоден, то есть выполняются некоторые необходимые требования для его использования. Вопрос заключается лишь в том, насколько данный алгоритм может быть векторизован и насколько велик может быть эффект от его использования по сравнению с другими используемыми подходами.

Особенности реализации алгоритма сортировки взаимодействий

Суть сортировки взаимодействий заключается в том, чтобы сортировать атомы двух смежных кубических ячеек (граничащих по грани, ребру или вершине) по проекции их координат на прямую, связывающую центры ячеек. Это позволяет гарантировать, что если в последовательности отсортированных таким образом атомов расстояние между проекциями координат окажется больше, чем радиус обрезания потенциала, $r > r_{cutoff}$, то для всех последующих атомов в последовательности также выполняется $r > r_{cutoff}$, и, следовательно, их можно отбросить. Данный подход существенно, от 16% до 59.4% [2], увеличивает в потоке данных долю подлежащих обработке атомов, что в свою очередь, дает ускорение работы LC в 2-2.5 раза.

Используемый в методе сортировки взаимодействий алгоритм быстрой сортировки qsort полностью себя оправдывает только для последовательного кода. При использовании векторизации удается поднять производительность многих этапов молекулярной динамики, но в силу того, что алгоритм qsort невекторизум, время сортировки при этом не изменяется. Для решения этой проблемы мы предлагаем применить алгоритм блочной сортировки.

Предложенный алгоритм блочной сортировки состоит из следующих частей (см. псевдокод алгоритма 0.1):

- расчет проекций координат каждого атома в ячейке;

Algorithm 0.1 Блочная сортировка взаимодействий

Require: Вход: массив проекций R. Выход: массив номеров атомов I
Require: Заданы минимальное R_{min} и максимальное R_{max} значения проекций атомов R[i]
Ensure: $R[I[i]] < R[I[j]]$ если $i < j$, где i, j - номера атомов
{1. Расчет проекций координат атомов}
for all номера атомов, i **do**
вычисление проекции R[i] координат атома i
end for
{2. Расчет номеров корзин}
for all номера атомов, i **do**
вычисление номера корзины k[i] по проекции R[i]
end for
{3. Упорядочение элементов}
{3.1. Расчет числа элементов в каждой корзине}
for all номера атомов, i **do**
offset[i] = left_bitwise_shift(1, k[i]) // перемещение бита элемента в позицию корзины
sum += offset[i] // суммирование смещенных битов элементов
end for
{3.2. Расчет конечных позиций корзин в выходном массиве I}
bucket = scan(sum)
{3.3. Расчет позиций атомов }
for all номера атомов, i **do**
bucket -= offset[i] // установка позиции записываемого элемента корзины
pos = right_bitwise_shift(bucket, k[i]) & 0xFF // извлечение позиции обработанного элемента
I[pos] = i; // запись сортированной позиции
end for
return Массив номеров I с порядком дальнейшей обработки атомов

- определение границ в пространстве, относительно которых решается принадлежность атомов той или иной корзине, и вычисление номеров корзин для каждого атома;
- переупорядочение атомов согласно их номерам корзин

Для последовательной версии (и без учета требований эффективности) реализация данного алгоритма тривиальна и не нуждается в обсуждении. Требования эффективности и векторизации алгоритма усложняют задачу.

Первая часть алгоритма - это расчет проекций r_n для всех частиц ячейки и заполнение массива проекций R. Каждая проекция r_n представляет собой скалярное произведение вектора координат частицы $\mathbf{r}-\mathbf{r}_0$, отсчитываемых от начала ячейки $\mathbf{r}_0$, на единичный вектор $\mathbf{n}$, связывающий центры соседних ячеек $r_n = (\mathbf{r}-\mathbf{r}_0, \mathbf{n})$. Поскольку все частицы лежат в одной ячейке, то данная операция может быть произведена в векторном режиме, например с помощью intrinsic функции _mm_dp_ps, доступной на процессорах с поддержкой SSE4. Число всевозможных направлений из центра ячейки на центры соседних ячеек равно 26, но в силу наличия симметрии (знаком направления можно пренебречь) расчеты проекций на каждом шаге динамики выполняется 13 раз. При использовании AVX (и в будущем AVX-512, AVX-1024) имеется возможность вычислять 2 и более проекций разных атомов за раз. Результатом выполнения данной части алгоритма является массив проекций R векторов координат атомов ячейки.

Вторая часть алгоритма - определение границ корзин и расчет номеров корзин для каждого атома ячейки. Для эффективности счета задается равное значение ширины w для всех корзин. При этом условии данная часть алгоритма элементарно векторизуется. Номер корзины определяется формулой $n_b = \lfloor r_p/w \rfloor$, где r_p - значение проекции координат атома, w - ширина корзины, а операция $\lfloor . \rfloor$ является операцией округления до ближайшего целого числа вниз. Однако следует учесть, что задание равной ширины корзин w приводит к существенной неравномерности заполнения корзин в зависимости от направления оси проектирования, относительно граней ячеек. Действительно, количество атомов dn_a, имеющих значение проекции r_p в узком слое dr_p зависит от площади $S(r_p)$ сечения ячейки перпендикулярном оси и проходящем через точку со значением r_p. Очевидно, что площадь сечения $S(r_p)$ существенно изменяется в зависимости от направления оси, и, в частности, она имеет максимум в центре ячейки, если ось связывает вершины куба. Дополнительной неравномерностью заполнения ячеек, обусловленной колебаниями локальной плотности атомов в пространстве, можно пренебречь. Результатом выполнения данной части алгоритма является массив, в котором каждому атому i соответствует номер корзины $k[i]$, в которую он попадает.

Третья часть алгоритма - определение порядка частиц. Переупорядочение самих частиц неэффективно, в силу того, что в реализации каждая частица представляется достаточно большой структурой, которая хранит множество атрибутов (координаты, заряд, параметры ван-дер-ваальсового взаимодействия, параметры связности и прочие). По этой причине вводится дополнительный массив I, в котором хранятся индексы частиц, и который собственно и определяет порядок их извлечения. Алгоритм создание такого массива I нельзя полностью векторизовать, поскольку в его основе лежит поэлементная запись по адресам в памяти. Тем не менее, некоторые шаги алгоритма могут быть реализованы таким образом, чтобы обрабатывать элементы совместно (аналог векторизации, или псевдовекторизации). Тривиальные решения, такие как создание отдельных массивов для каждой корзины с последующими копированиями результатов в конечный массив, не рассматривались в силу очевидной неэффективности.

Совместная обработка возможна при упаковке данных в одно длинное беззнаковое целое число. К примеру, длинное 8-байтовое целое long long int способно хранить данные для 8 корзин, при условии, что под каждую корзину выделяется 1 байт. Учитывая ограничение нашей задачи, что число частиц в ячейке не превышает 128 элементов (при нормальной плотности и при размере ребра ячейки в 10Åчисло атомов в ячейке составляет порядка 90), следует что возможностей long long int вполне достаточно для хранения всех нужных данных. С использованием упаковки данных третья часть алгоритма выглядит следующим образом:

- На шаге 3.1 алгоритма определяется количество элементов, попадающих в каждую корзину. Для этого мы проходим по массиву, хранящему номера корзин для всех элементов и добавляем 1 к той части числа sum, которая соответствует номеру корзины.
- На шаге 3.2 алгоритма, зная число элементов во всех корзинах, определяются позиции конца записи для всех корзин. Поскольку гарантируется, что каждая следующая корзина содержит элементы, строго большие, чем предыдущая, то позиция конца любой корзины будет равна сумме числа элементов всех предыдущих корзин с числом элементов в текущей корзине, то есть позиции концов корзин задаются последовательностью: $\{n_0, n_0 + n_1, n_0 + n_1 + n_2..\}$, где n_i - число элементов в корзине i. Данная задача решается с помощью аналога операции scan [16], и для 8 корзин ее решение требует всего 3 операции сдвига (на 8, 16 и 32 бита) с суммированием (на рис. 1 показаны первые 2 шага алгоритма).
- На шаге 3.3 алгоритма вычисляются позиции упорядочения каждого элемента. Для этого, перед каждой записью элемента позиция записи сдвигается на 1 элемент вниз, а после каждый элемент записывается по текущей позиции корзины. Массив I хранит конечный порядок элементов. Данный этап алгоритма невекторизуем.

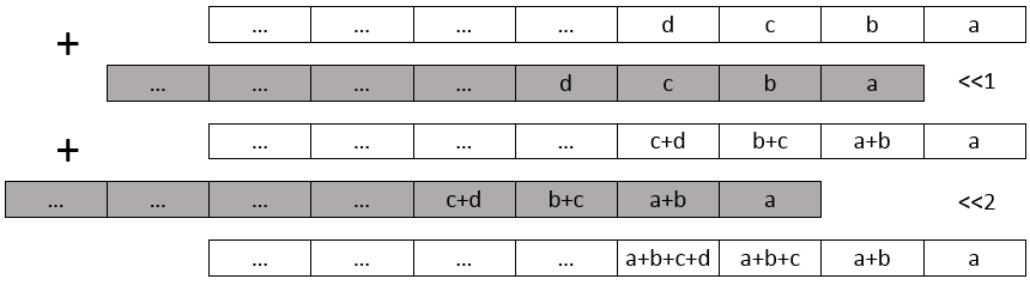

Рис. 1. Выполнение операции scan над числом, хранящим количество элементов в корзинах

Предложенный алгоритм по сравнению с другими быстрыми алгоритмами сортировки допускает векторизацию (и псевдовекторизацию) почти всех своих подшагов, что является важным преимуществом. Недостатком такой векторизованной сортировки является то, что количество обрабатываемых атомов в векторном режиме должно быть кратно длине вектора. По этой причине в исходную последовательность атомов приходится включать так называемые «dummy» или «пустые» атомы, с которыми взаимодействия не рассчитываются.

Результаты

Влияние типа сортировки

Алгоритм блочной сортировки в LC + IS был реализован в рамках программного комплекса МОЛКЕРН [17] (C++, многопоточность на основе библиотеки boost). В качестве объекта тестирования был выбран одноатомный газ - аргон. Выбор обусловлен тем, что для одноатомной системы возможно проверить работу алгоритма и эффект от его векторизации более явно, если снизить побочное влияние множества других алгоритмов молекулярной динамики (например, избежать учета времени работы алгоритмов, отслеживающих связность атомов в молекулах). Замеры времени выполнялись путем накопления времени работы алгоритма счетчиком clock_gettime, входящим в стандарт POSIX-20013, и имеющим точность в несколько наносекунд. Все замеры времени усреднялись по 10 запускам с удалением 3-х наиболее худших результатов, обусловленных побочным влиянием на время выполнения программы посторонних процессов операционной системы.

В качестве тестового случая использовалась система из 512 (8*8*8) ячеек, заполненных 17525 атомами аргона, время динамики составляло 5 пс. Сборка осуществлялась с уровнем оптимизации -O2 при помощи компилятора gcc 4.6.3. Для учета влияния различных архитектур сравнение проводилось на 2 процессорах - AMD Phenom II X4 945 (3.6 Ггц, микроархитектура K10) и Intel Core i5 450M (2.4 Ггц, микроахритектура Nehalem).

Таблица 1. Затраты времени на сортировку и исполнение (в секундах) в зависимости от типа процессора (AMD и Intel), метода сортировки (qsort - метод LC + IS, bsort - метод LC + BS) и наличия SSE-векторизации. Ускорение приведено относительно qsort в невекторизованном режиме.

метод	время сортировки/общее(сек)		доля сортировки(%)		ускорение сортировки/общее	
	AMD	**Intel**	**AMD**	**Intel**	**AMD**	**Intel**
qsort	122.4/764.0	137.5/1005	16%	13.6%	-	-
qsort + SSE	125.0/645.0	136.9/780.7	19.4%	17.5%	0.97/1.18	1.00/1.28
bsort	97.7/783.0	90.7/1009	12.5%	9%	1.25/0.98	1.52/0.99
bsort + SSE	79.3/595.0	65.1/707	13.3%	9.2%	1.54/1.28	2.11/1.42

В таблице 1 показано время работы алгоритма сортировки (быстрая сортировка qsort для LC + IS метода и блочная сортировка bsort для метода LC + BS) без и с SSE-векторизацией для разных типов процессоров, а также полное время работы программы (полное время сортировки и полное время работы программы разделены слешем). Также в таблице представлены данные, касающиеся того, какую долю от полного времени выполнения занимает сортировка, и какое ускорение достигается при использовании блочной (bsort, с/без SSE-векторизации) и быстрой (qsort+SSE) сортировок относительно невекторизованного кода для LC + IS метода (qsort), как для ускорения самого этапа сортировок, так и для полного ускорения выполнения программы (данные также разделены слешем).

Как видно из таблицы, доля этапа сортировок в общем времени выполнения программы в исходном невекторизованном коде метода LC + IS достаточно высока, от 13.6% до 16%. При векторизации программы доля данного этапа возрастает от 17.5% до 19.4%, что обусловлено, как ранее было указано, тем, что используемый алгоритм сортировок qsort невекторизуем. Изменение алгоритма сортировок на блочную сразу приводит к заметному уменьшению времени ее выполнения, в 1.25 и 1.52 раза. Однако видно, что несмотря на то, что блочная сортировка работает быстрее в последовательном варианте, выигрыша по общей производительности не наблюдается. Это объясняется необходимостью обработки дополнительных элементов корзин (dummy элементов) при расчетах. SSE векторизация кода заметно уменьшает время выполнения программы от 1.18 до 1.42 раза для AMD K10 и Nehalem. Причем видно, что в любом случае корзинная сортировка дает лучшие результаты, чем быстрая сортировка, а ее векторизация дает дополнительный прирост производительности, который отсутствует для быстрой сортировки. Из представленных данных видно, что в последовательном варианте блочная сортировка работает в 1.25 раз быстрее на процессоре AMD и в 1.5 раза быстрее на процессоре Intel, а при использовании SSE векторизации разница возрастает до 1.58 раз на AMD и до 2.1 раз на процессоре Intel. В целом это приводит к увеличению скорости работы программы в SSE-векторизованном варианте относительно SSE-векторизованного кода, использующего qsort, на 10% для процессора AMD и на 14% для процессора Intel. Меньший выигрыш на процессоре AMD вероятно обусловлен меньшей эффективностью векторизации на данной архитектуре. Что же касается разницы в абсолютных значениях замера времен выполнения, то она связана с различной частотой использованных процессоров.

Влияние на масштабируемость

Масштабируемость программы в зависимости от изменения числа потоков является одной из наиболее важных ее характеристик, поскольку показывает, насколько эффективно задействуется каждое вычислительное ядро процессора. В идеальном случае при каждом удвоении активных ядер, время работы программы должно также двукратно сокращаться. Как отмечено в [11], и косвенным образом видно из зависимостей времени работы алгоритмов от числа потоков [9] и [10], масштабируемость современных методов расчета несвязующих взаимодействий (IVT, LC, LC + IS, LVT, PVT) далеко неидеальна и требует серьезного рассмотрения. В данной статье не обсуждается зависимость масштабирования от используемой машинной архитектуры процессоров, хотя влияние архитектуры, естественно, является определяющим.

Масштабируемость реализаций алгоритмов в зависимости от изменения числа потоков выполнения тестировались на кластере ЦКП «Биоинформатика» (НКС-30Т) [18]. Для расчетов использовался 1 узел кластера, включающий в себя 2 процессора Intel Xeon X5670 (тактовая частота 2.93 Ггц, 6 ядер). Для сборки программы использовался компилятор gcc-4.1. Параметры расчетов были следующими: 4096 (16*16*16) ячеек с длиной ребра 10.6Å, ≈ 300000 атомов аргона, 10 шагов оптимизации, время динамики - 10 пс. Каждый метод запускался с числом потоков от 1 до 12.

Таблица 2. Сравнение времени исполнения SSE-реализации (в секундах) в зависимости от числа потоков и ускорение метода LS + BS по сравнению с методами LS + IS и LVT. Цифры над значениями времени показывают уровень масштабирования относительно выполнения расчетов в однопоточном режиме.

Число потоков	LC+IS	LVT	LC+BS	LC+BS vs LC+IS	LC+BS vs LVT
1	12233	14606	11237	9%	30%
2	$8161^{1.5}$	$8499^{1.72}$	$6224^{1.8}$	31%	36%
4	$5653^{2.16}$	$4981^{2.93}$	$4018^{2.8}$	40%	24%
6	$5072^{2.41}$	$4045^{3.61}$	$3545^{3.17}$	43%	14%
12	$3886^{3.14}$	$3016^{4.84}$	$2664^{4.22}$	45%	13%

В таблице 2 представлены данные по времени исполнения (в секундах) короткой 10 пс молекулярной динамики в зависимости от числа потоков для различных методов расчета, реализованных в программе MOLKERN: LC + IS - метод связанных ячеек с сортировкой взаимодействий алгоритмом qsort, LC + BS - метод связанных ячеек с сортировкой взаимодействий алгоритмом блочной сортировки, LVT - метод локальных Верлет таблиц. Цифры над значениями времени показывают уровень масштабирования реализации метода относительно выполнения в однопоточном режиме. Также в таблице даны цифры, показывающие во сколько раз метод LS + BS превосходит метод локальных таблиц LVT, который является лучшим среди последних предложенных методов, связанных с наличием промежуточного объекта для хранения данных - Верлет таблицы. Во всех методах была использована SSE-векторизация. На рис. 2 эти же данные приведены в графическом виде.

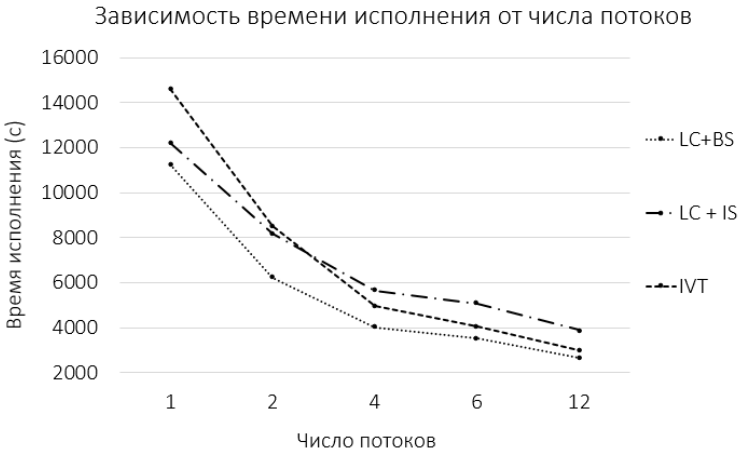

Рис. 2. Сравнительный график зависимости времени исполнения от числа потоков

Как видно из представленных выше данных, уровень масштабирования всех методов неидеален и качественно в целом совпадает у разных методов. Он является наихудшим для метода LC + IS и наилучшим для LVT. Последнее легко объяснимо учитывая тот факт, что метод LVT был специально предложен для решения проблемы масштабирования на многоядерных процессорах с разделяемой памятью. Для метода LC + BS уровень масштабирования имеет промежуточное значение. Что же касается эффективности, то лучшим по времени выполнения во всем диапазоне числа потоков является метод LC + BS с используемой для сортировки взаимодействий алгоритма блочной сортировки. В целом видно, что предложенный нами метод обходит лучший из имеющихся методов, связанных с наличием промежуточного объекта для хранения данных - Верлет таблицы, метод LVT, на величину от 13% до 30%.

Заключение

В рамках данной работы предложено для сортировки взаимодействий в методе LC + IS использовать алгоритм, основанный на алгоритме блочной сортировки. Показано, что предложенный алгоритм ликвидирует основной недостаток метода сортировки взаимодействий, связанный с увеличением накладных расходов на сортировку при переходе к векторизации. Данный подход обеспечивает лучшую производительность как последовательной, так и векторизованной версии, позволяя снизить накладные расходы на сортировки от 1.2 до 2.1 раз, что в целом увеличивает эффективность выполнения молекулярной динамики данным методом в диапазоне от 10% до 14% в однопоточном выполнении и до 45% в многопоточном. Данный подход также имеет преимущество от 13% до 30% над лучшим в настоящее время вариантом метода Верлет таблицы - методом локальных Верлет таблиц, хотя и уступает последнему по уровню масштабирования.

Данная работа поддержана грантом СО РАН: интеграционный проект №130.

Список литературы

[1] Quentrec B. ; Brot C. //*J. Comput. Phys.* 1973, 13, 430.

[2] Gonnet P. //*J. Comput. Chem.*, 28, 2, 2007, с. 570-573.

[3] Verlet L. //*Phys Rev* 1967, 159, 98.

[4] Meloni S., Rosati M. // *J. Chem. Phys.* 2007, 126, 121102.

[5] Fomin E. //*J. Comput. Chem.*, 33, 1, 2012, с. 76–81

[6] Maximova, T.; Keasar, C. //*J. Comput. Biol.* 2006, 13, 1041.

[7] Yao, Z.; Wang, J.-S.; Liu, G.-R.; Cheng, //*M. Comput Phys Commun* 2004, 161, 27.

[8] Cui, Z. W.; Sun, Y.; Qu, J. M. Chin //*Sci Bull* 2009, 54, 1463.

[9] Gonnet P. //*J Comput Chem.* 2012 Jan 5;33(1):76-81

[10] Gonnet P. ECS Technical Report 2013/01.

[11] Фомин Э.С., Алемасов Н.А., Матвиенко С.А. //*Материалы X Международной конференции*, г.Пермь, 1-3 ноября, 2010 г., Т.2, с.284-291.

[12] Intel® Manycore Testing Lab http://software.intel.com/en-us/intel-manycore-testing-lab

[13] Фомин Э. С. //*Вычислительные методы и программирование*, 2010, Т.11, 299-305.

[14] Hoare C.A.R. Quicksort //*The Computer Journal*, 1962, 5 (1): 10-16.

[15] Knuth D. E. The Art of Computer Programming. Vol. 3. Sorting and Searching, *Addison-Wesley*, 1973.

[16] Blelloch G. E. Scans as primitive parallel operations. // *Computers, IEEE Transactions on*, 1989, 38(11), 1526 - 1538.

[17] Фомин Э.С. , Алемасов Н.А., Чирцов А.С.,Фомин. А.Э. //*Биофизика*, 51, 7, 2006, с.110-113.

[18] Гибридный кластер НКС-30Т+GPU http://www2.sscc.ru/Information/NEW/Inform-SSCC.htm