DOI: https://doi.org/10.30898/1684-1719.2026.10.10
УДК: 004.94:004.272

# Экспериментальное исследование масштабируемости параллельных реализаций алгоритма молекулярной динамики с короткодействующим потенциалом Леннарда-Джонса на Python/MPI, C++/MPI и MATLAB/parfor

**Д.Н. Забегаев, П.С. Глазунов, В.А. Вдовин**

**ИРЭ им. В.А. Котельникова РАН,**
**Россия, Москва**

Статья поступила в редакцию 30 октября 2026 г.

**Аннотация.** В работе выполнено сравнение эффективности реализаций параллельных вычислений при решении задачи молекулярной динамики на языках программирования Python и C++, а также в среде MATLAB. Рассматривается классическая задача моделирования движения системы взаимодействующих частиц в трехмерной области пространства. Используется потенциал взаимодействия Леннарда-Джонса. Для интегрирования уравнений движения применяется скоростной метод Верле. Для всех исследуемых программных платформ реализованы идентичные алгоритмы вычислений и одинаковые начальные условия моделирования. Параллельные вычисления в MATLAB организованы с использованием оператора parfor, обеспечивающего распараллеливание на уровне потоков (threads), а в программах на Python и C++ – с применением стандарта передачи сообщений MPI. Проведены численные эксперименты на двух вычислительных платформах: персональном компьютере и многопроцессорном сервере. Исследованы зависимости времени выполнения программ от количества вычислительных процессов. Выполнен

сравнительный анализ производительности, масштабируемости и эффективности распараллеливания. Показано, что реализация в среде MATLAB обеспечивает наименьшее абсолютное время выполнения вычислений, тогда как программа на языке C++ демонстрирует наилучшую масштабируемость при увеличении числа процессов. Полученные результаты могут быть использованы при выборе программных средств для разработки программ молекулярно-динамического моделирования.

**Ключевые слова:** молекулярная динамика, параллельные вычисления, MPI, MATLAB, Python; C++, скоростной метод Верле, высокопроизводительные вычисления, численный эксперимент, моделирование физических процессов.

**Финансирование:** Работа выполнена по государственному заданию ИРЭ им. В.А. Котельникова РАН.

**Автор для переписки:** Владимир Александрович Вдовин, vdv@cplire.ru

## Введение

Молекулярная динамика (МД) представляет собой метод компьютерного моделирования, основанный на численном решении уравнений движения для систем взаимодействующих частиц. Данный подход позволяет исследовать структуру, динамику и термодинамические свойства вещества на элементарном уровне, а также анализировать механизмы процессов, недоступных для прямого экспериментального наблюдения. Метод МД является одним из наиболее распространенных методов компьютерного моделирования атомных систем [1-6]. К числу наиболее передовых областей применения МД относятся задача нахождения белков для биологических рецепторов [7-9], оптимизация химических реакций в электролитах аккумуляторов [10-12], моделирование катализа и динамики поверхностных химических реакций [13-15], исследования движения дислокаций и структурных дефектов в твердых телах [16-18].

В последние годы методы МД получают новый импульс развития, связанный с совершенствованием массово-параллельных вычислительных систем и переходом к сопряженной CPU-GPU архитектуре [19-20]. В связи

с этим исследование степени масштабируемости параллельных МД вычислений, выполненных в различных средах разработки, представляется актуальным.

Как правило, для решения задач МД используют уже готовые хорошо известные программные пакеты, такие как GROMACS, LAMMPS, NAMD и другие. Возможности к многопоточному распараллеливанию некоторых из них изучались ранее в работах [21-23].

Однако вышеупомянутые пакеты могут не подходить для решения некоторых нестандартных задач, таких, например, где потенциал взаимодействия частиц зависит от случайных событий. Так, после поглощения фотона молекула может переходить между несколькими энергетическими состояниями, что сопровождается скачкообразным изменением потенциала этой молекулы. Это нужно учитывать при моделировании систем с фотохимией [24-25] и систем, находящиеся под лазерным или рентгеновским облучением [26]. Иногда сложность возникает за счет необходимости использования определенных граничных условий, в частности, задания «плавающего» потенциала на электродах [27]. Наконец, отметим, что алгоритмы решения уравнений движения в GROMACS, LAMMPS и NAMD не приспособлены для разрывных потенциалов, что сужает область их применимости [28]. Кроме того, для методов моделирования редких событий (например, для нерешеточного кинетического метода Монте-Карло) вообще еще не сформировался единый широко распространенный универсальный программный комплекс: существующие программные пакеты являются узкоспециализированными и используют различные алгоритмические подходы [29-31].

В таких случаях возникает необходимость разработки собственных программных средств для МД моделирования, что, в свою очередь, ставит задачу выбора оптимального программного инструмента. Для решения подобных задач широко применяются языки программирования Python, C++ и вычислительная среда MATLAB, отличающиеся как вычислительной эффективностью, так и удобством разработки, возможностями расширения и поддержки высокопроизводительных вычислений. В связи с этим представляет интерес

сравнительное исследование указанных инструментов с точки зрения их применимости к задачам МД.

## 1. Постановка задачи

Целью настоящей работы является экспериментальное сравнение эффективности параллельной реализации одного и того же алгоритма молекулярной динамики (с короткодействующим потенциалом Леннарда-Джонса), разработанного с применением Python (MPI), C++ (MPI) и MATLAB (Parallel Computing Toolbox, parfor). При этом используются одинаковая физическая модель, начальные условия, алгоритм интегрирования и вычислительная схема, что обеспечивает корректность сравнения. В работе исследуются зависимость времени выполнения программы от числа процессов, время выполнения программы на одном вычислительном потоке, минимальное время выполнения программы и соответствующее ему оптимальное число потоков, а также максимальное достигаемое ускорение (относительно последовательного запуска).

Рассматривается задача моделирования динамики системы взаимодействующих частиц методами классической МД. Основная идея метода заключается в численном решении уравнений движения для большого числа частиц, взаимодействующих друг с другом посредством заданного потенциала.

В рамках МД подхода каждая частица рассматривается в качестве материальной точки, движение которой описывается законами классической механики. Мы будем предполагать, что в начальный момент времени частицы случайным образом распределены в расчетной области трехмерного пространства. Для каждой частицы задаются начальные скорости, характеризующиеся фиксированным значением модуля и случайным направлением движения.

Движение каждой частицы определяется вторым законом Ньютона

$$m_i \frac{d^2 \vec{r}_i}{dt^2} = \overrightarrow{F_i}, \tag{1}$$

где $m_i$ – масса $i$-й частицы, $\vec{r}_i$ – ее радиус-вектор, $\overrightarrow{F_i}$ – суммарная сила, действующая на частицу со стороны всех остальных частиц системы.

Силы взаимодействия между частицами определяются как отрицательный градиент потенциальной энергии системы $U$

$$\vec{F} = -\vec{\nabla} U. \tag{2}$$

В качестве модели взаимодействия в работе используется потенциал Леннарда-Джонса [32]

$$U_{\mathrm{LJ}}(r) = \varepsilon \left( \left( \frac{r_{\min}}{r} \right)^{12} - 2 \left( \frac{r_{\min}}{r} \right)^{6} \right), \tag{3}$$

где $r$ – расстояние между двумя частицами, $\varepsilon$ – глубина потенциальной ямы, $r_{\min}$ – расстояние между частицами, соответствующее минимуму потенциальной энергии. Данный потенциал качественно описывает как короткодействующее отталкивание частиц при малых расстояниях, так и притяжение на больших расстояниях. Качественный график зависимости $U_{\mathrm{LJ}}(r)$ приведен на рис. 1.

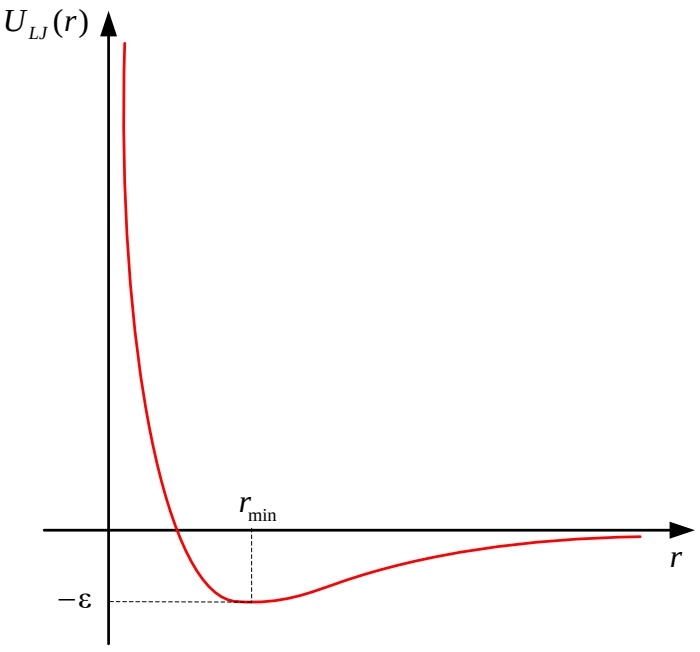


Рис. 1. Качественный график зависимости потенциала Леннарда-Джонса от расстояния между двумя частицами.

Особенностью потенциала Леннарда-Джонса является его быстрое убывание при увеличении расстояния между частицами. В связи с этим при численном моделировании целесообразно использовать радиус отсечения взаимодействия $R_{\text{off}}$, при превышении которого взаимодействием между частицами можно пренебречь (рис. 2).

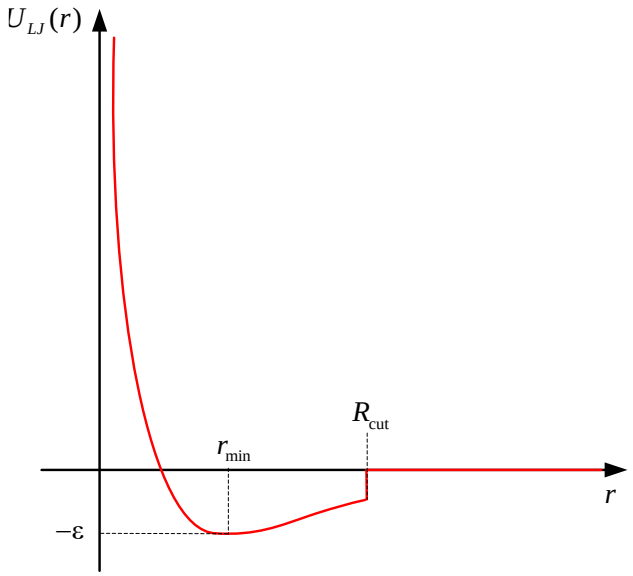


Рис. 2. Качественный график зависимости усеченного потенциала Леннарда-Джонса от расстояния между двумя частицами.

Однако такое простое отсечение потенциала приводит к разрыву его производной, что вызывает затруднение при ее численном расчете. Для устранения данной проблемы используется сглаживающая функция, обеспечивающая непрерывность потенциала в области отсечения. Пример такой функции приведен ниже (формула (4)), а ее график показан на рис. 3.

$$w(r)=\begin{cases}1, r \le R_{\text{on}} \\ \dfrac{(R_{\text{off}}^2 - r^2)^2\left(R_{\text{off}}^2 - 3R_{\text{on}}^2 + 2r^2\right)}{(R_{\text{off}}^2 - R_{\text{on}}^2)^3}, R_{\text{on}} < r < R_{\text{off}} \\ 0, r > R_{\text{off}}\end{cases}. \quad (4)$$

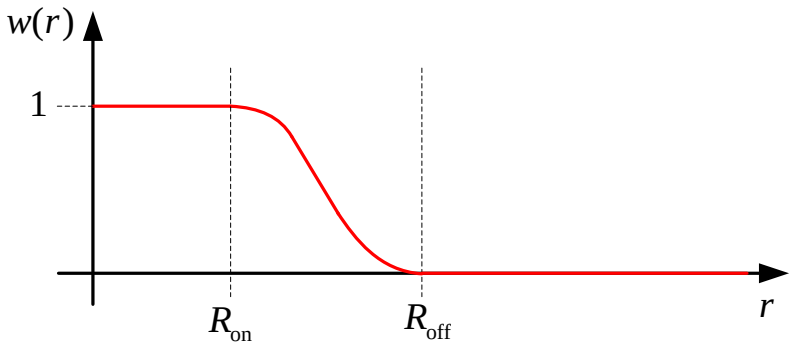


Рис. 3. График сглаживающей отсекающей функции $w(r)$.

Функция $w(r)$ сконструирована таким образом, что ее производные обращаются в ноль при $r = R_{off}$, и при $r = R_{on}$

$$\left.\frac{dw(r)}{dr}\right|_{r=R_{off}} = \left.\frac{dw(r)}{dr}\right|_{r=R_{on}} = 0. \quad (5)$$

В результате в данной работе для расчетов использовался «усеченный» сглаженный потенциал Леннарда-Джонса

$$U_{LJ-cut}(r) = \varepsilon w(r)\left(\left(\frac{r_{min}}{r}\right)^{12} - 2\left(\frac{r_{min}}{r}\right)^{6}\right) \quad (6)$$

с $R_{on} = 2r_{min}$ и $R_{off} = 2{,}5r_{min}$ (согласно рекомендациям [3], приведенным в §2.2 и §3.3). Соответствующие графики изображены на рис. 4.

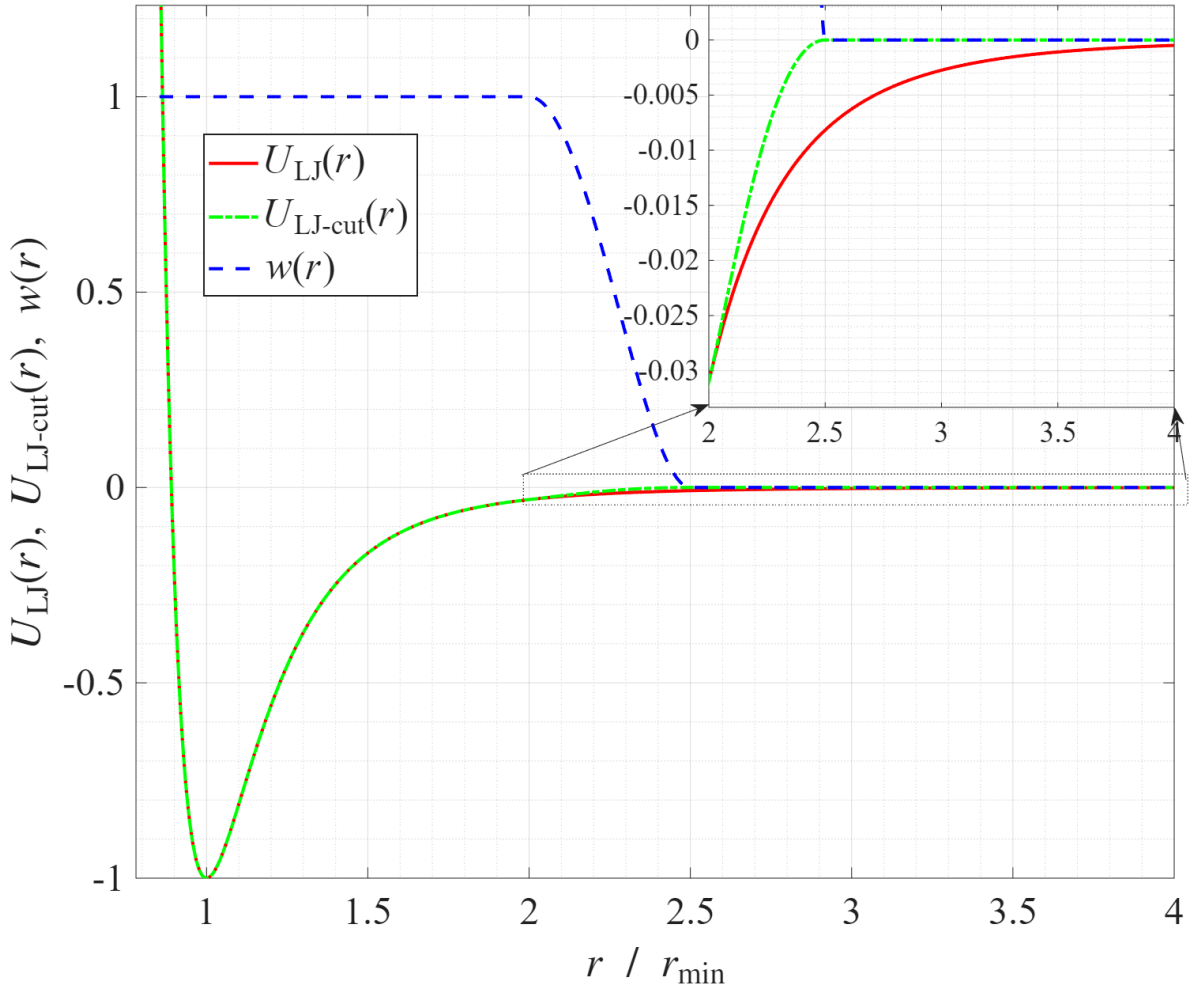


Рис. 4. Графики зависимостей сглаживающей функции $w(r)$, потенциала Леннарда-Джонса $U_{LJ}(r)$ и усеченного потенциала Леннарда-Джонса $U_{LJ\text{-}cut}(r)$ от расстояния между частицами $r$ при $r_{min} = 1$, $\varepsilon = 1$, $R_{on} = 2r_{min}$, $R_{off} = 2{,}5r_{min}$.

Полная потенциальная энергия системы определяется суммой энергий всех частиц

$$U = \frac{1}{2}\sum_{\substack{i=1,\ldots,N \\ j=1,\ldots,N \\ i\neq j}} U\left(r_{ij}\right) = \sum_{1\le j<i\le N} U\left(r_{ij}\right), \tag{7}$$

где $N$ – число частиц системы, $r_{ij}$ – расстояние между частицами с номерами $i$ и $j$.

Тогда ускорения частиц определяются через производные потенциальной энергии

$$a_{x,i} = -\frac{1}{Cm_i}\frac{\partial U}{\partial x_i}, a_{y,i} = -\frac{1}{Cm_i}\frac{\partial U}{\partial y_i}, a_{z,i} = -\frac{1}{Cm_i}\frac{\partial U}{\partial z_i}, \tag{8}$$

где $C$ – нормировочная константа, возникающая при переходе к системе единиц, удобной для МД моделирования. В настоящей работе используются единицы, в которых масса измеряется в атомных единицах, расстояние – в ангстремах, время – в пикосекундах, а энергия – в электрон-вольтах. При таком выборе оказывается, что $C$ = 9648,5.

Численное решение уравнений движения осуществляется с использованием скоростного метода Верле [33]. На каждом временном шаге выполняется последовательное обновление координат, вычисление новых ускорений и обновление скоростей частиц.

Одной из основных вычислительных проблем молекулярной динамики является высокая вычислительная сложность расчета взаимодействий. При прямом вычислении взаимодействий между всеми парами $N$ частиц количество операций пропорционально $N^2$, что приводит к значительному увеличению времени расчетов при больших размерах системы.

Для уменьшения вычислительных затрат в работе используется метод списка ячеек, подразумевающий пространственное разбиение расчетной области на ячейки. Каждая частица относится к определенной ячейке в зависимости от значений своих координат. При вычислении потенциала, в котором находится выбранная частица, учитываются только частицы, расположенные в той же ячейке или соседних ячейках. Поскольку радиус взаимодействия ограничен величиной $R_{\text{off}}$, такой способ позволяет существенно сократить количество проверяемых пар частиц без потери точности расчетов. Подобный подход широко используется в современных пакетах молекулярной динамики для уменьшения вычислительной сложности расчетов [34].

## 2. Описание работы программ

Были разработаны три программные реализации решения задачи молекулярной динамики средствами языков программирования Python, C++ и MATLAB. Все программы решают одинаковую вычислительную задачу и реализуют идентичный алгоритм моделирования. Это обеспечивает

корректность последующего сравнения вычислительной эффективности этих программных инструментов.

Разработанные программы моделируют динамику системы взаимодействующих частиц. Расчеты выполняются в трехмерной области пространства размером 200×200×200 Å. В рамках численного эксперимента моделируется система, состоящая из $10^4$ частиц аргона с атомной массой 39,938 а.е.м.

На первом этапе работы программы формируются начальные условия системы. Частицы случайным образом распределяются внутри расчетной области пространства. При генерации координат контролируется минимально допустимое расстояние между частицами, равное $r_{\text{lim}} = 3{,}82$ Å, что предотвращает возникновение чрезмерно больших сил отталкивания между частицами. В случае невозможности размещения частицы после заданного числа попыток процесс генерации завершается досрочно. После задания координат каждой частице присваивается начальная скорость $V = 3$ Å/пс. Направление скорости определяется случайным образом.

Глубина потенциальной ямы задаваемого потенциала Леннарда-Джонса составляет $\varepsilon = 0{,}01$, а характерное расстояние взаимодействия определяется параметром $r_{\text{min}} = 3{,}822$ Å. В расчетах применяется радиус отсечения взаимодействия $R_{\text{off}} = 2.5 r_{\text{min}} = 9{,}7$ Å. Размер ячейки при разбиении области пространства выбирался равным 10 Å, при этом все трехмерное пространство разбивается на 20×20×20 = 8000 ячеек.

В программе реализованы граничные условия в виде жестких стенок. При достижении частицей границы расчетной области нормальная к поверхности стенки компонента скорости изменяет знак, что соответствует абсолютно упругому отражению частицы от стенки.

Шаг интегрирования по времени составляет $dt = 0{,}01$ пс. Моделирование выполняется в течение 20 временных шагов в рамках одного вычислительного цикла. В процессе расчетов производится измерение времени выполнения

программы для последующего анализа вычислительной эффективности реализации.

## 3. Реализация параллельных вычислений в MATLAB, Python и C++

Для организации параллельных вычислений в среде MATLAB (версия R2022b на портативном компьютере и версия R2025b на сервере) использовался оператор parfor из пакета Parallel Computing Toolbox. Распараллеливание производится на уровне потоков (threads). Данный механизм позволяет автоматически распределять независимые итерации цикла между несколькими вычислительными потоками. В рассматриваемой реализации распараллеливание применялось на этапах вычисления ускорений частиц и обработки граничных условий. Поскольку вычисления для каждой частицы на данных этапах не зависят от результатов вычислений для других частиц, итерации соответствующих циклов могут выполняться независимо друг от друга, что делает возможным их распараллеливание средствами parfor.

Стоит отметить, что использование parfor позволяет сократить время выполнения вычислений без существенного изменения структуры последовательного алгоритма. При запуске цикла MATLAB автоматически выполняет распределение вычислительной нагрузки между заданным количеством рабочих процессов, синхронизацию данных и объединение результатов после завершения вычислений. Благодаря этому достигается упрощение разработки параллельной программы по сравнению с более низкоуровневыми средствами организации параллельных вычислений.

Реализация программы на языке Python выполнялась с использованием стандарта передачи сообщений MPI (Message Passing Interface) [35]. Для организации параллельных вычислений применялась библиотека mpi4py, предоставляющая интерфейс MPI для языка программирования Python. Разработка программного кода осуществлялась в интегрированной среде PyCharm.

Запуск программы непосредственно из среды PyCharm может выполняться только в последовательном режиме, то есть, на одном логическом процессоре. Для организации параллельных вычислений использовалась внешняя утилита mpiexec версии 4.0.3, входящая в состав реализации Microsoft MPI 10.0.12498.5, а также установленный в операционной системе интерпретатор Python версии 3.9.6.

Параллельный запуск программы осуществлялся из командной строки с использованием команды

*mpiexec -n q python program_name.py*,

где *q* – количество вычислительных процессов, а *program_name.py* – имя файла кода программы.

Реализация программы на языке C++ выполнялась также с использованием стандарта передачи сообщений MPI. Разработка программного кода осуществлялась в интегрированной среде разработки Microsoft Visual Studio 2022. Компиляция производилась с помощью Microsoft Visual C++ (MSVC) 19.36.32534 с ключом оптимизации -O2 для повышения производительности исполняемой программы. Для организации параллельных вычислений использовался заголовочный файл *mpi.h*, предоставляющий интерфейс Microsoft MPI для языка C++.

Для параллельного выполнения программы в операционной системе Windows был установлен пакет MS-MPI, обеспечивающий поддержку распределенных вычислений и взаимодействия между вычислительными процессами.

Параллельный запуск программы осуществлялся из командной строки операционной системы с использованием команды

*mpiexec -n q program_name.exe*,

где *q* – количество вычислительных процессов, а *program_name.exe* – имя исполняемого файла скомпилированной программы.

Кроме того, после соответствующей настройки параметров среды разработки запуск параллельной программы мог выполняться непосредственно из среды Visual Studio.

В реализациях на Python и C++ распараллеливание осуществлялось посредством распределения множества частиц между вычислительными процессами MPI. Каждый процесс выполнял вычисления для выделенной ему группы частиц, включая расчет ускорений и обновление параметров движения. После завершения локальных вычислений производился обмен данными между процессами, обеспечивающий согласованность глобального состояния системы.

## 4. Результаты вычислительных экспериментов

Для оценки эффективности различных средств организации параллельных вычислений были проведены вычислительные эксперименты на двух вычислительных платформах: на портативном компьютере на базе 4-ядерного процессора AMD Ryzen 7 3700U с 8 логическими потоками обработки данных, с 12 Гб оперативной памяти, работающем под управлением операционной системы Windows 10 и на высокопроизводительном сервере на базе 28-ядерного процессора Intel Xeon Gold 6330 с 56 логическими потоками обработки данных и 1024 Гб оперативной памяти, работающем под управлением операционной системы Windows Server (Таблица 1).

Таблица 1. Характеристики вычислительных платформ.

| **Параметр** | **Ноутбук** | **Сервер** |
|---|---|---|
| Процессор | AMD Ryzen 7 | Intel Xeon |
| Количество ядер процессора | 4 | 28 |
| Количество вычислительных потоков | 8 | 56 |
| Оперативная память | 12 ГБ | 1024 ГБ |
| Операционная система | Windows 10 | Windows |

На рис. 5 представлены результаты измерения времени выполнения программ на языках Python и C++, а также в среде MATLAB на портативном компьютере при изменении количества вычислительных процессов от 1 до 8. Синими точками обозначены экспериментально измеренные значения времени выполнения, а красными линиями – результаты аппроксимации экспериментальных данных универсальным законом масштабируемости Нила Гюнтера [36].

Как видно, экспериментальные данные достаточно точно аппроксимируются законом Гюнтера, представленном в виде зависимости времени выполнения программы $T$ от количества используемых для расчета процессов $N$. Данный закон можно записать в следующем виде

$$T(N) = T_1\left[\alpha + \frac{1-\alpha}{N} + \beta(N-1)\right], \tag{9}$$

здесь $T_1$ – время выполнения программы в случае, когда для расчетов используется всего один процесс (то есть $T_1 \equiv T(1)$); $\alpha$ – доля операций в программе, которые (по тем или иным причинам) не могут быть выполнены параллельно; $\beta$ – коэффициент, характеризующий потери производительности, обусловленные взаимодействием (отправлением и чтением сообщений) между процессами.

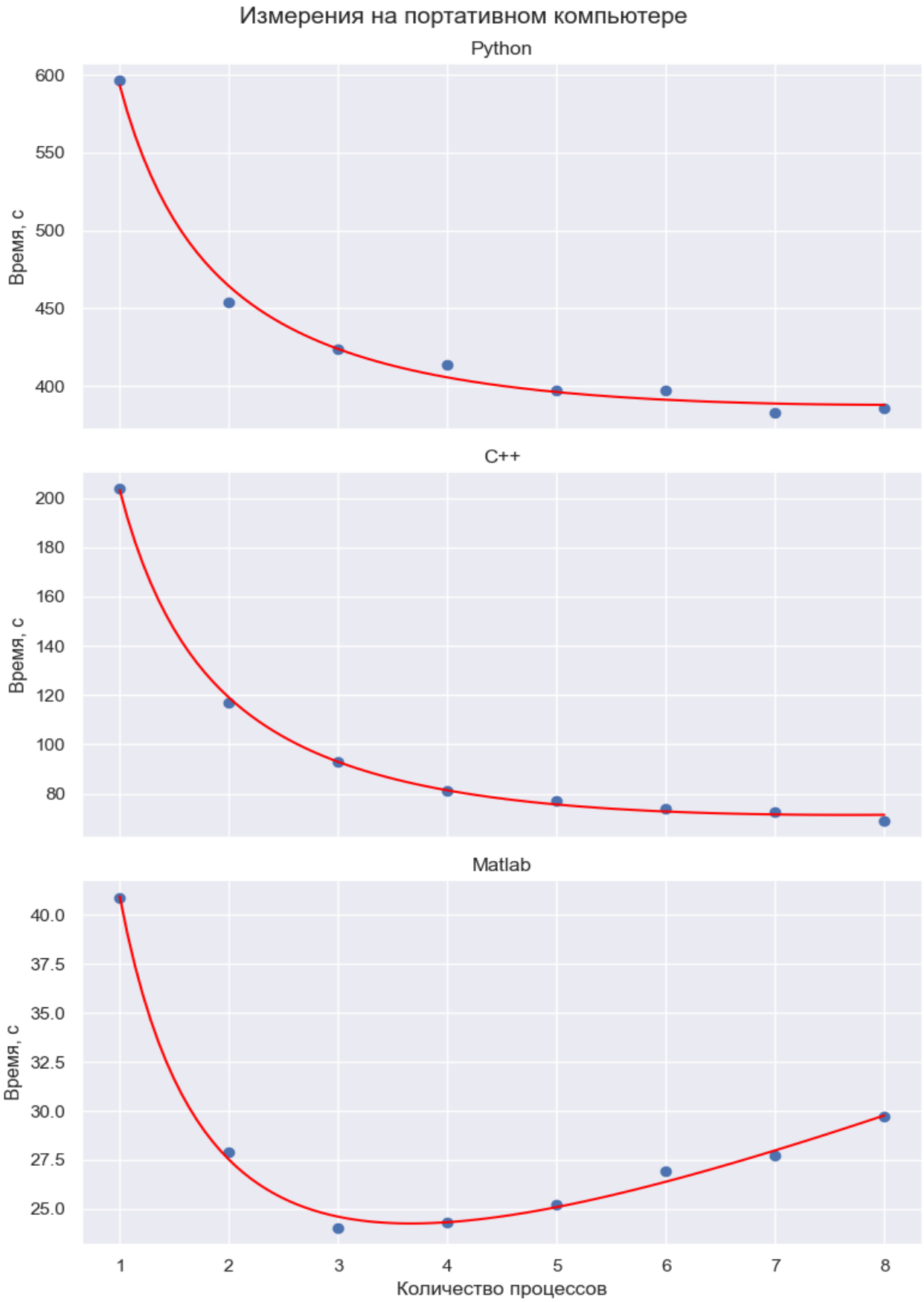


Рис. 5. Зависимость времени выполнения программ на Python, C++ и MATLAB от количества процессов на портативном компьютере.

В форме закона Гюнтера (9) хорошо выделяются три составляющие времени выполнения программы $T(N)$.

- $T_1\alpha$ – составляющая, которая в рамках модели не уменьшается с ростом числа процессов. Она может отражать последовательные вычисления, обусловленные самим алгоритмом расчета или же конкуренцию процессов за общие ресурсы и другие ограничения масштабируемости.
- $T_1\frac{1-\alpha}{N}$ – составляющая, которая в идеальном случае уменьшается обратно пропорционально числу процессов.
- $T_1\beta(N-1)$ – дополнительные расходы, возрастающие с увеличением числа процессов: обмен данными, синхронизация, координация.

Очевидно, что при добавлении одного процесса время $T_1\beta(N-1)$ увеличивается на величину $\beta T_1$. Поэтому $\beta$ можно интерпретировать как долю времени $T_1$, которое добавляется ко времени обмена данными, синхронизацию и координацию между процессами при добавлении одного нового процесса. По этой причине коэффициенты $\alpha$ и $\beta$ удобно выражать в процентах.

Аппроксимация экспериментальных кривых, представленных на рис. 5, уравнением (9) с помощью метода наименьших квадратов дает следующие значения коэффициентов: для измерений на языке Python: $T_1$ = 593 с, $\alpha$ = 55,2%, $\beta$ = 0,65%; для измерений на языке C++: $T_1$ = 203 с, $\alpha$ = 14,2%, $\beta$ = 1,46%; для измерений в Matlab: $T_1$ = 41 с, $\alpha$ = 22,9%, $\beta$ = 5,74%.

Из рис. 5 следует, что во всех случаях увеличение количества процессов приводит к уменьшению времени выполнения программы на начальном участке зависимости. Наибольшее абсолютное время выполнения наблюдается для реализации на языке Python, промежуточные значения демонстрирует программа на языке C++, а минимальное время выполнения характерно для реализации в MATLAB.

Для более наглядного сравнения полученных результатов на рис. 6 представлены сглаженные зависимости времени выполнения всех трех программ на одном графике.

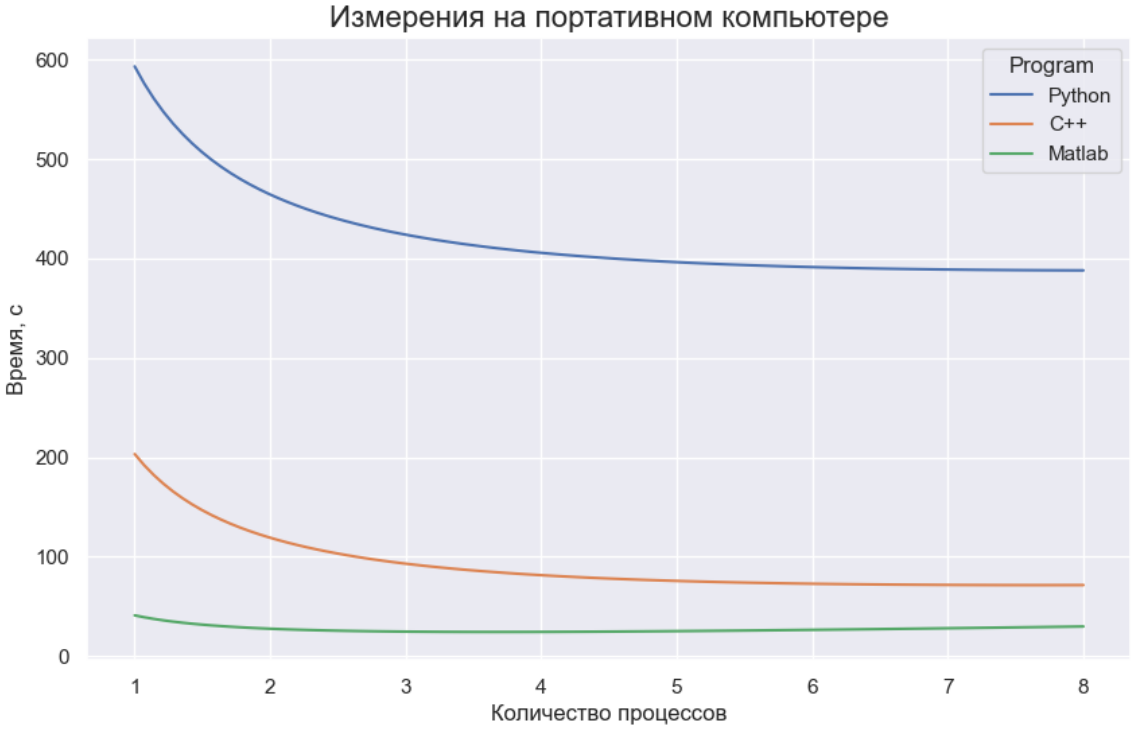


Рис. 6. Сравнение времени выполнения программ на Python, C++ и MATLAB на портативном компьютере.

Из рис. 6 видно, что реализация на языке Python остается наиболее медленной во всем исследованном диапазоне числа процессов. Реализация на языке C++ обеспечивает более высокую производительность, тогда как программа MATLAB демонстрирует наименьшее время выполнения вычислений.

Аналогичные эксперименты были проведены на сервере с числом процессов от 1 до 56. Результаты измерений представлены на рис. 7. Синими точками показаны экспериментальные данные, красными линиями – их аппроксимация при помощи закона Гюнтера (9). Значения коэффициентов, получаемых при этой аппроксимации с помощью метода наименьших квадратов: для измерений на языке Python $T_1$ = 233 с, $\alpha$ = 57,13%, $\beta$ = 0,023%; для измерений на языке C++ $T_1$ = 215 с, $\alpha$ = 4,35%, $\beta$ = 0,190%; для измерений в среде Matlab $T_1$ = 42 с, $\alpha$ = 7,68%, $\beta$ = 0,78%.

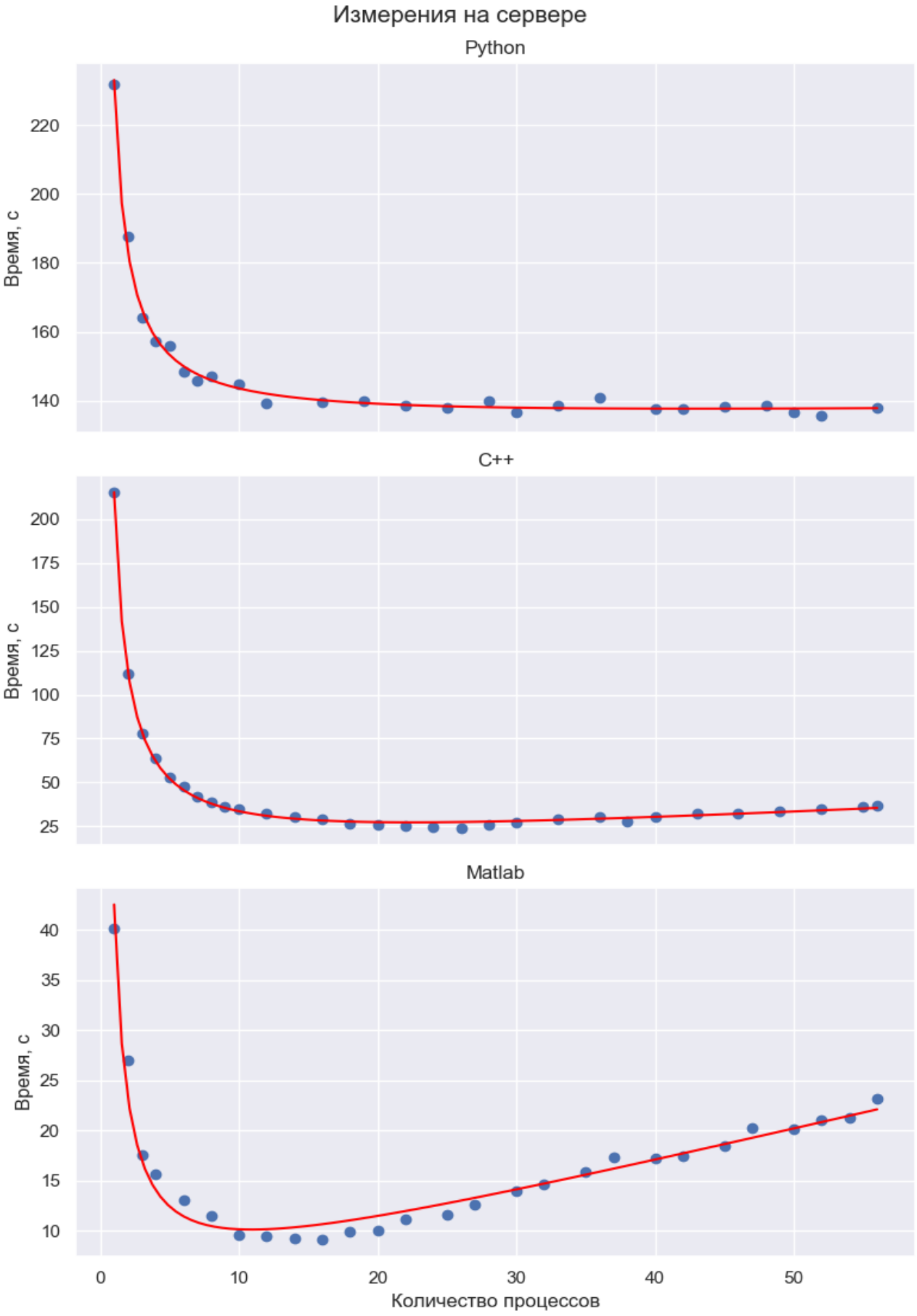


Рис. 7. Зависимость времени выполнения программ на Python, C++ и MATLAB от количества процессов на сервере.

Полученные зависимости демонстрируют существенное уменьшение времени выполнения при увеличении числа процессов на начальном участке. Для программ на языке C++ и в среде MATLAB после достижения некоторого минимального значения наблюдается последующее увеличение времени выполнения. Для реализации на языке Python после первоначального снижения времени выполнения зависимость выходит на участок, близкий к горизонтальному.

Сравнение закона Гюнтера для всех трех программ представлено на рис. 8. Из него следует, что на серверной платформе минимальное время выполнения также достигается для реализации в среде MATLAB. Реализация на языке C++ занимает промежуточное положение, а программа на языке Python характеризуется наибольшим временем выполнения во всем диапазоне исследованных значений.

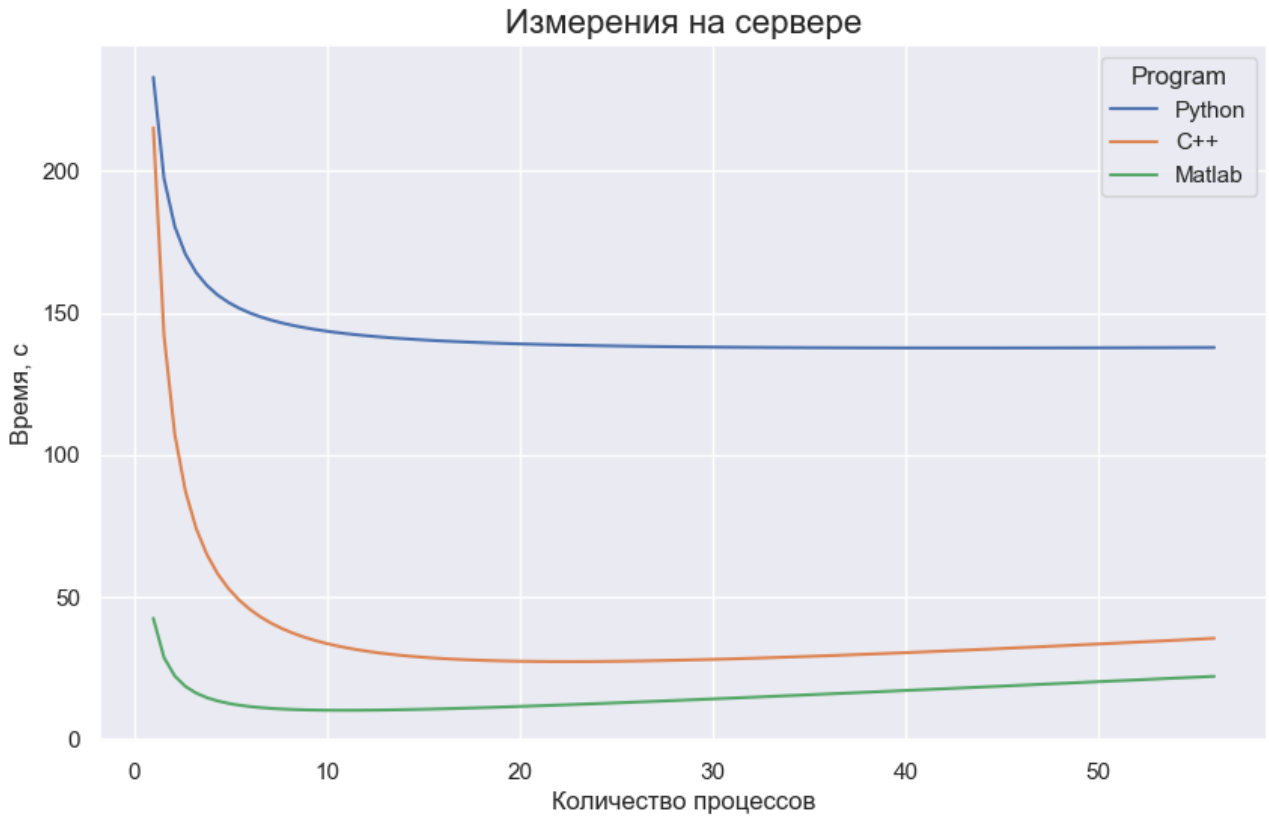


Рис. 8. Сравнение времени выполнения программ на Python, C++ и MATLAB на сервере.

## 5. Обсуждение результатов

Для количественного сравнения различных реализаций были определены время выполнения программы на одном вычислительном потоке $T_1$, минимальное время выполнения программы $T_{\min}$, соответствующее ему оптимальное число потоков $p_{\text{опт}}$ и достигаемое ускорение относительно последовательного запуска $S = T_1/T_{\min}$. Результаты представлены в Таблице 2.

Полученные результаты показывают, что увеличение числа вычислительных процессов не приводит к пропорциональному сокращению времени выполнения программы. Для всех исследованных реализаций наблюдается быстрое уменьшение времени вычислений при переходе от одного процесса к нескольким процессам. Однако после достижения некоторого оптимального значения дальнейшее увеличение числа процессов либо практически не влияет на производительность, либо приводит к ее ухудшению.

Таблица 2. Основные характеристики исследуемых программных реализаций.

| **Платформа** | $T_1$, с | $T_{\min}$, с | $p_{\text{опт}}$ | Ускорение $S = T_1/T_{\min}$ |
|---|---|---|---|---|
| Python (ноутбук) | 596,3 | 383,0 | 7 | 1,56 |
| C++ (ноутбук) | 203,9 | 69,1 | 8 | 2,95 |
| MATLAB (ноутбук) | 40,9 | 24,0 | 3 | 1,70 |
| Python (сервер) | 231,7 | 136,7 | 25 | 1,69 |
| C++ (сервер) | 215,0 | 24,2 | 26 | 8,88 |
| MATLAB (сервер) | 40,1 | 9,1 | 16 | 4,41 |

Такое поведение соответствует закону Амдала [37], согласно которому ускорение параллельной программы ограничивается наличием последовательной части алгоритма. По мере увеличения числа процессов вклад последовательных участков программы становится доминирующим, вследствие чего эффективность дальнейшего распараллеливания уменьшается. Закон

Амдала является частным случаем более общего закона универсальной масштабируемости Нила Гюнтера, если в последнем положить $\beta = 0$.

Дополнительным фактором являются накладные расходы, связанные с организацией параллельных вычислений. В случае реализации на основе MPI к таким расходам относятся обмен данными между процессами, синхронизация вычислений и передача результатов промежуточных расчетов. При малом числе процессов выигрыш от распараллеливания существенно превышает указанные затраты. Однако при дальнейшем увеличении числа процессов объем межпоточного взаимодействия возрастает, вследствие чего эффективность распараллеливания снижается.

При сравнении вычислений, проведенных на сервере и ноутбуке, можно выделить несколько закономерностей.

1) Для C++ и MATLAB переход на сервер сопровождается значительным уменьшением обоих коэффициентов $\alpha$ и $\beta$. Это свидетельствует о лучшей масштабируемости реализаций алгоритма при переходе на сервер.
2) Для Python коэффициент $\alpha$ практически одинаков на обеих машинах – примерно 0,55-0,57. Это означает, что существенная часть алгоритма программы плохо поддается ускорению независимо от количества доступных ядер.
3) Коэффициент $\beta$ во всех случаях значительно меньше на сервере. При этом величина уменьшения различается: примерно в 28 раз у Python, в 7,7 раза у C++ и в 7,4 раза у MATLAB.

Вероятно, эти различия объясняются тем, как процессы взаимодействуют с памятью и друг с другом. Дело в том, что каждый процесс должен выполнять вычисления, используя некоторый общий массив данных. Поэтому при увеличении числа процессов возникает конкуренция за пропускную способность памяти, кэш, вычислительные блоки процессора, как говорят, возникает «бутылочное горлышко». На ноутбуке – обычном персональном компьютере – даже с четырьмя физическими ядрами конкуренция процессов может проявиться уже при относительно небольшом их количестве. На сервере – машине,

специально оптимизированной для параллельных вычислений – эта нагрузка, вероятно, способна более равномерно распределяться между имеющимся количеством аппаратных ресурсов.

Кроме того, имеют значение различия в размерах и устройстве кэш-памяти, пропускной способности памяти, частоте процессора и ее изменении под нагрузкой. Стоит заметить, что увеличение количества ядер само по себе не обязано уменьшать коэффициент $\alpha$. Например, если 20% времени программы занимает строго последовательный участок ее кода, то добавление ядер его не ускорит. Уменьшение коэффициента $\alpha$ означает, что изменилось относительное влияние плохо масштабируемых участков и аппаратных ограничений. Это может объяснить уменьшение $\alpha$ на сервере для C++ и MATLAB. Что касается коэффициента $\beta$, вероятно, его уменьшение связано со следующими обстоятельствами. При использовании MPI разные процессы имеют собственные адресные пространства. Поэтому для согласования данных им приходится выполнять обмены информацией (при работе на одном компьютере они могут осуществляться через общую память, без передачи по сети). Стоимость этих операций зависит от архитектуры памяти, количества доступных ядер, размещения процессов, скорости копирования данных и поведения операционной системы. В MATLAB применяется другая модель распараллеливания – parfor с пулом потоков (threads). Это позволяет работать с общей памятью, но не устраняет накладные расходы на распределение итераций и синхронизацию. Поэтому уменьшение $\beta$ на сервере можно связать с тем, что относительные затраты на взаимодействие и координацию процессов оказываются меньше.

При сравнении вычислений, проведенных в различных средах программирования, можно выделить следующие особенности:

- Python: большое значение коэффициента $\alpha$, но небольшое значение $\beta$;
- C++: небольшое $\alpha$ и умеренное $\beta$;
- MATLAB: небольшое $\alpha$, но относительно большое $\beta$.

Для Python коэффициент $\alpha$ составляет примерно 55–57% на обеих машинах. Если положить $\beta = 0$, то при неограниченном увеличении количества процессов $N \to \infty$ время выполнения будет стремиться к $\alpha T_1$. Следовательно, модель Гюнтера для Python предсказывает сохранение более половины исходного времени выполнения, даже если не учитывать время обмена информацией между процессами. Возможная причина этого заключается в особенностях выполнения Python-кода. Операции интерпретатора, динамическая типизация, создание объектов и работа со структурами данных могут существенно влиять на время выполнения программы. Вероятно, в реализации Python имеются достаточно дорогие вычислительные участки или операции подготовки и обмена данными библиотеки mpi4py, которые не получают пропорционального ускорения при увеличении числа процессов.

Реализация C++ на сервере демонстрирует наименьшее значение $\alpha = 4{,}35\%$. Это согласуется с наблюдаемым хорошим масштабированием: на серверной платформе время выполнения уменьшается приблизительно от 214 с до 24 с, что соответствует ускорению почти в девять раз. При этом $\beta = 0{,}19\%$ – ненулевое, поэтому после определенного значения $N$ расходы на организацию параллельных вычислений начинают перевешивать выигрыш от распределения нагрузки. Возможно, в C++ реализуются более компактные структуры данных, меньшие накладные расходы на обработку отдельных частиц и более эффективное выполнение компилированных циклов.

MATLAB показывает минимальное абсолютное время выполнения, но его коэффициент $\beta$ больше, чем у Python и C++. MATLAB может очень быстро выполнять отдельные вычислительные операции благодаря оптимизированным встроенным функциям и JIT-компиляции. Однако применение parfor сопровождается внутренними затратами среды MATLAB на управление пулом рабочих процессов и синхронизацию вычислений. Таким образом, MATLAB одновременно демонстрирует самое маленькое время расчета и сравнительно худшую масштабируемость.

Исходные коды программ, использованных в работе, доступны в репозитории GitHub [38].

## Заключение

В работе выполнено воспроизводимое сравнение трех параллельных программных реализаций одной и той же задачи молекулярной динамики для систем с потенциалом Леннарда-Джонса, разработанных на языке программирования C++, языке программирования Python и в вычислительной среде MATLAB. Во всех случаях использовались одинаковые математическая модель, алгоритм вычислений, начальные условия и параметры моделирования, что позволило провести сопоставление производительности различных программных платформ.

Для организации параллельных вычислений в среде MATLAB использовался оператор parfor, а в программах на языках Python и C++ применялся стандарт передачи сообщений MPI. Численные эксперименты проводились на двух платформах, отличающихся вычислительной мощностью и количеством доступных процессорных ядер.

Полученные результаты показали, что увеличение количества вычислительных процессов приводит к существенному сокращению времени выполнения программы лишь до некоторого оптимального значения. Дальнейшее увеличение числа процессов сопровождается снижением эффективности распараллеливания вследствие роста накладных расходов на синхронизацию и обмен данными между процессами, а также наличием последовательной части алгоритма, не поддающейся распараллеливанию.

Установлено, что реализация в среде MATLAB обеспечивает наименьшее абсолютное время выполнения вычислений на обеих исследованных платформах. При этом программа на языке C++ демонстрирует наилучшую масштабируемость и наиболее эффективно использует увеличение числа вычислительных потоков. Реализация на языке Python характеризуется наибольшим временем выполнения и меньшей эффективностью

распараллеливания по сравнению с двумя другими вариантами. Вместе с тем использование языка Python также может быть оправдано благодаря большему удобству разработки, высокой читаемости программного кода и наличию широкого набора специализированных библиотек для обработки научных данных.

Дополнительным практическим критерием сравнения является объем исходного кода: реализации на Python, C++ и MATLAB содержат соответственно 654, 407 и 421 строку программного кода без учета пустых строк и комментариев. При этом Python и C++ являются свободно распространяемыми средствами, тогда как MATLAB – коммерческая система, требующая лицензии.

Полученные результаты подтверждают существенное влияние выбора программной платформы и технологии организации параллельных вычислений на производительность программ молекулярно-динамического моделирования. Результаты работы могут быть использованы при разработке высокопроизводительных программных комплексов для решения задач вычислительной физики и молекулярной динамики. Работа представляет воспроизводимые экспериментальные данные и практические ориентиры по выбору инструмента для решения задач аналогичного класса.

**Финансирование:** Работа выполнена по государственному заданию ИРЭ им. В.А. Котельникова РАН.

## Литература

1. Allen M.P., Tildesley DJ.. Computer simulation of liquids. – Oxford university press, 2017. https://doi.org/10.1093/oso/9780198803195.001.0001
2. Frenkel D., Smit B., Ratner MA.. Understanding molecular simulation: from algorithms to applications. – 1997. https://doi.org/10.1016/B978-0-12-267351-1.X5000-7
3. Рапапорт Д. Искусство молекулярной динамики. – 2012.

4. Панькин Н.А., Смоланов Н.А., Кашапов Н.Ф. Введение в метод классической молекулярной динамики. – Казань: Издательство Казанского университета, 2022. – 156 с.
5. Хельтье Х.Д. и др. Молекулярное моделирование. Теория и практика. – М.: Бином. Лаб. Знаний. – 2013.
6. Норман Г.Э., Стегайлов В.В. Стохастическая теория метода классической молекулярной динамики // Математическое моделирование. – 2012. – Т. 24. – № 6. – С. 3–44.
7. Verma V.A. et al. Utilizing Molecular Dynamics and Mechanistic Pharmacokinetic Studies in the Design of Selective CDK2 Inhibitors // Journal of Medicinal Chemistry. – 2026. – V. 69. – No. 13. – P. 15294–15316. https://doi.org/10.1021/acs.jmedchem.5c03803
8. Gallego G.M. et al. Discovery of Atirmociclib (PF-07220060): A Potent and Selective CDK4 Inhibitor // Journal of Medicinal Chemistry. – 2025. – V. 68. – No. 24. – P. 26085–26098. https://doi.org/10.1021/acs.jmedchem.5c02137
9. Elez K. et al. Simulations and active learning enable efficient identification of an experimentally-validated broad coronavirus inhibitor // Nature Communications. – 2025. – V. 16. – P. 6949. https://doi.org/10.1038/s41467-025-62139-5
10. Chang X. et al. Integrating Molecular Dynamics and Machine Learning for Solvation-Guided Electrolyte Optimization in Lithium Metal Batteries // Advanced Science. – 2025. – V. 12. – No. 36. – P. e04997. https://doi.org/10.1002/advs.202504997
11. Yu Z. et al. Simulation guided molecular design of hydrofluoroether solvent for high energy batteries // Journal of Materials Chemistry A. – 2024. – V. 12. – P. 6294–6301. https://doi.org/10.1039/D3TA07670A
12. Li G. et al. Multilevel Artificial Intelligent Framework Accelerates Electrolytes Design for Aqueous Batteries // Angewandte Chemie International Edition. – 2026. – V. 65. – No. 20. – P. e5593105. https://doi.org/10.1002/anie.5593105

13. Chen X. et al. Tuning the Hydrogen Bond Network Inside the Helmholtz Plane for Industrial Hydrogen Evolution // Advanced Functional Materials. – 2025. – V. 35. – No. 36. – P. 2503701. https://doi.org/10.1002/adfm.202503701
14. Chen W. et al. Designer topological-single-atom catalysts with site-specific selectivity // Nature Communications. – 2025. – V. 16. – P. 574. https://doi.org/10.1038/s41467-025-55838-6
15. Tian F. et al. Bioinspired tandem catalyst with topological ordered water networks accelerates alkaline hydrogen evolution electrocatalysis // Applied Catalysis B: Environment and Energy. – 2026. – V. 386. – P. 126427. https://doi.org/10.1016/j.apcatb.2026.126427
16. Yang Y. et al. Rejuvenation as the origin of planar defects in the CrCoNi medium entropy alloy // Nature Communications. – 2024. – V. 15. – P. 1402. https://doi.org/10.1038/s41467-024-45696-z
17. Xie B. et al. Divergent evolution of slip banding in CrCoNi alloys // Nature Communications. – 2025. – V. 16. – P. 3631. https://doi.org/10.1038/s41467-025-58480-4
18. Lyu S. et al. Statistical mechanics, entropy and temperature analog of dislocations moving on fluctuating resistance landscapes // Acta Materialia. – 2025. – V. 291. – P. 121002. https://doi.org/10.1016/j.actamat.2025.121002
19. Desai S., Reeve S.T., Belak J.F. Implementing a neural network interatomic model with performance portability for emerging exascale architectures // Computer Physics Communications. – 2022. – V. 270. – P. 108156. https://doi.org/10.1016/j.cpc.2021.108156
20. Beck T.L., Carloni P., Asthagiri D.N. All-atom biomolecular simulation in the exascale Era // Journal of chemical theory and computation. – 2024. – V. 20. – No. 5. – P. 1777-1782. https://doi.org/10.1021/acs.jctc.3c01276
21. Kutzner C. et al. Scaling of the GROMACS Molecular Dynamics Code to 65k CPU Cores on an HPC Cluster // Journal of Computational Chemistry. – 2025. – V. 46. – No. 5. – P. e70059. https://doi.org/10.1002/jcc.70059

22. Phillips J.C. et al. Scalable molecular dynamics on CPU and GPU architectures with NAMD // The Journal of chemical physics. – 2020. – V. 153. – No. 4. https://doi.org/10.1063/5.0014475
23. Abraham M.J. et al. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers // SoftwareX. – 2015. – V. 1–2. – P. 19–25. https://doi.org/10.1016/j.softx.2015.06.001
24. Mai S., Marquetand P., González L. Nonadiabatic dynamics: The SHARC approach // WIREs Computational Molecular Science. – 2018. – V. 8. – No. 6. – P. e1370. https://doi.org/10.1002/wcms.1370
25. Shu Y. et al. Semiclassical Nonadiabatic Molecular Dynamics Using Linearized Pair-Density Functional Theory // Journal of Chemical Theory and Computation. – 2024. – V. 20. – No. 20. – P. 8741–8748. https://doi.org/10.1021/acs.jctc.4c01061
26. Medvedev N., Artímez Peña A. Ultrafast X-ray induced damage and nonthermal melting in cadmium sulfide // Physical Chemistry Chemical Physics. – 2025. – V. 27. – No. 16. – P. 8230–8237. https://doi.org/10.1039/D5CP00525F
27. Coretti A. et al. MetalWalls: Simulating electrochemical interfaces between polarizable electrolytes and metallic electrodes // The Journal of Chemical Physics. – 2022. – V. 157. – No. 18. https://doi.org/10.1063/5.0101777
28. Bannerman M.N., Sargant R., Lue L. DynamO: a free O(N) general event-driven molecular dynamics simulator // Journal of Computational Chemistry. – 2011. – V. 32. – No. 15. – P. 3329–3338. https://doi.org/10.1002/jcc.21915
29. Ntioudis S., Ewen J.P., Dini D., Turner C.H. PAPRECA: A parallel hybrid off-lattice kinetic Monte Carlo/molecular dynamics simulator // Journal of Open Source Software. – 2024. – V. 9. – No. 98. – P. 6714. https://doi.org/10.21105/joss.06714
30. Béland L.K. et al. Kinetic activation–relaxation technique and self-evolving atomistic kinetic Monte Carlo: Comparison of on-the-fly kinetic Monte Carlo algorithms // Computational Materials Science. – 2015. – V. 100. – P. 124-134. https://doi.org/10.1016/j.commatsci.2014.12.001

31. Hayakawa S. et al. Atomistic modeling of meso-timescale processes with SEAKMC: A perspective and recent developments // Computational Materials Science. – 2021. – V. 194. – P. 110390. https://doi.org/10.1016/j.commatsci.2021.110390
32. Jones J.E. On the determination of molecular fields. II. From the equation of state of a gas // Proceedings of the Royal Society of London. Series A. – 1924. – V. 106. – No. 738. – P. 463–477. https://doi.org/10.1098/rspa.1924.0082
33. Verlet L. Computer “experiments” on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules // Physical review. – 1967. – V. 159. – No. 1. – P. 98. https://doi.org/10.1103/PhysRev.159.98
34. Plimpton S. Fast Parallel Algorithms for Short-Range Molecular Dynamics // Journal of Computational Physics. – 1995. – V. 117. – No. 1. – P. 1–19. https://doi.org/10.1006/jcph.1995.1039
35. Gropp W., Lusk E., Skjellum A. Using MPI: Portable Parallel Programming with the Message-Passing Interface. 3rd ed. Cambridge, MA: MIT Press, 2014. 336 p.
36. Gunther N.J. Guerrilla capacity planning: a tactical approach to planning for highly scalable applications and services. – Berlin, Heidelberg : Springer Berlin Heidelberg, 2007. https://doi.org/10.1007/978-3-540-31010-5
37. Amdahl G.M. Validity of the single processor approach to achieving large scale computing capabilities // Proceedings of the April 18–20, 1967, Spring Joint Computer Conference (AFIPS ’67 Spring). – 1967. – V. 30. – P. 483–485. https://doi.org/10.1145/1465482.1465560
38. md-scalability [Электронный ресурс]. GitHub repository. Режим доступа: https://github.com/DmitriyZabegaev/md-scalability

**Для цитирования:**

Забегаев Д.Н., Глазунов П.С., Вдовин В.А., Экспериментальное исследование масштабируемости параллельных реализаций алгоритма молекулярной динамики с короткодействующим потенциалом Леннарда–Джонса на Python/MPI, C++/MPI и MATLAB/parfor // Журнал радиоэлектроники. – 2026. – № 10. https://doi.org/10.30898/1684-1719.2026.10.10