



doi 10.26089/NumMet.v27r435

УДК 519.6,
519.85,
004.42

Анализ параллельной сложности и масштабируемости OpenMP реализаций матричной модификации метода муравьиных колоний для параметрической оптимизации

Ю. П. Титов

Московский авиационный институт (национальный исследовательский университет),
Москва, Российская Федерация

ORCID: 0000-0002-9093-6755, e-mail: kalengul@mail.ru

К. А. Кордовер

Московский авиационный институт (национальный исследовательский университет),
Москва, Российская Федерация

ORCID: 0009-0009-0545-3555, e-mail: k.a.kordover@mail.ru

П. А. Жданов

Московский авиационный институт (национальный исследовательский университет),
Москва, Российская Федерация

ORCID: 0009-0003-3898-5466, e-mail: p.a.zhdanov@mail.ru

А. А. Жданов

Московский авиационный институт (национальный исследовательский университет),
Москва, Российская Федерация

ORCID: 0009-0003-0950-7828, e-mail: a.a.zhdanov@mail.ru

Аннотация: Статья посвящена разработке и анализу матричной модификации метода муравьиных колоний для задач параметрической оптимизации большой размерности. Теоретически доказано снижение асимптотической сложности с $O(K \cdot n^2)$ для канонического метода муравьиных колоний до $O(K \cdot n)$ для матричной модификации, где n — число параметров, K — размер популяции. На основе анализа арифметической интенсивности строго обоснована принадлежность алгоритма к классу memory-bound с пределом $\nu \approx 0.44$ оп/байт. Экспериментально на 7 процессорах различных архитектур достигнуто ускорение до 10.7 на 18 потоках. Выявлены четыре режима масштабируемости, определяемые иерархией кэш-памяти. Для гибридной архитектуры Intel Alder Lake определены паттерны масштабирования для P- и E-ядер. Предложена блочная модификация алгоритма, обеспечивающая практически линейное ускорение за счет управления L2-резидентностью приватных буферов. Разработаны аналитические соотношения для выбора оптимального числа блоков.

Ключевые слова: метод муравьиных колоний, параметрическая оптимизация, матричный алгоритм, параллельные вычисления, OpenMP, memory-bound, ложное разделение кэша, иерархия памяти, масштабируемость.

Благодарности: Работа выполнена в рамках государственного задания Минобрнауки России, проект № FSFF-2026-0008.

Для цитирования: Титов Ю.П., Кордовер К.А., Жданов П.А., Жданов А.А. Анализ параллельной сложности и масштабируемости OpenMP реализаций матричной модификации метода муравьиных колоний для параметрической оптимизации // Вычислительные методы и программирование. 2026. 27, № 4. 559–584. doi 10.26089/NumMet.v27r435.



Analysis of parallel complexity and scalability of OpenMP implementations of the matrix modification of the ant colony method for parametric optimization

Yurii P. Titov

Moscow Aviation Institute (National Research University),
Moscow, Russia

ORCID: 0000-0002-9093-6755, e-mail: kalengul@mail.ru

Konstantin A. Kordover

Moscow Aviation Institute (National Research University),
Moscow, Russia

ORCID: 0009-0009-0545-3555, e-mail: k.a.kordover@mail.ru

Pavel A. Zhdanov

Moscow Aviation Institute (National Research University),
Moscow, Russia

ORCID: 0009-0003-3898-5466, e-mail: p.a.zhdanov@mail.ru

Alexander A. Zhdanov

Moscow Aviation Institute (National Research University),
Moscow, Russia

ORCID: 0009-0003-0950-7828, e-mail: a.a.zhdanov@mail.ru

Abstract: This paper is devoted to the development and analysis of a matrix modification of the ant colony optimization method (Matrix-ACO) for high-dimensional parametric optimization problems. A reduction in asymptotic complexity from $O(K \cdot n^2)$ (canonical ACO) to $O(K \cdot n)$, where n is the number of parameters and K is the population size, is theoretically proven. Based on the analysis of arithmetic intensity, the algorithm's belonging to the memory-bound class with a limit of $\nu \approx 0.44$ ops/byte is rigorously justified. An acceleration of up to 10.7 times on 18 threads was experimentally achieved on 7 processors of various architectures. Four scalability modes determined by the cache hierarchy are identified. For the Intel Alder Lake hybrid architecture, scaling patterns for P- and E-cores are determined. A block-based modification of the algorithm is proposed, providing nearly linear speedup by controlling the L2 residency of private buffers. Analytical relationships for selecting the optimal number of blocks are developed.

Keywords: ant colony optimization, parametric optimization, matrix algorithm, parallel computing, OpenMP, memory-bound, false sharing, memory hierarchy, scalability.

Acknowledgements: The work was carried out within the framework of the state assignment of the Ministry of Science and Higher Education of the Russian Federation, project No. FSFF-2026-0008.

For citation: Yu. P. Titov, K. A. Kordover, P. A. Zhdanov and A. A. Zhdanov, "Analysis of parallel complexity and scalability of OpenMP implementations of the matrix modification of the ant colony method for parametric optimization," Numerical Methods and Programming. 27 (4), 559–584 (2026). doi 10.26089/NumMet.v27r435.

1. Введение. Метод муравьиных колоний (Ant Colony Optimization, ACO), впервые предложенный в начале 1990-х годов как Ant Algorithms [1] и развитый до Ant System [2], является одним из наиболее успешных метаэвристических алгоритмов, инспирированных природными системами. Классический ACO использует программных агентов ("муравьев"), каждый из которых строит решение, оставляя феромонные следы на пройденном пути. За десятилетия развития было предложено множество модификаций, направленных на решение проблемы стагнации и улучшение скорости сходимости: Elitist Ant System (EAS) [3], ANT-Q [4], Ant System на основе ранга (RAS) [5], MAX-MIN Ant System (MMAS) [6], Best-Worst Ant System (BWAS) [7] и др.

Исторически сложилось так, что подавляющее большинство реализаций ACO и его параллельных модификаций сфокусированы на решении комбинаторных задач, в первую очередь задачи ком-



мивояжера (TSP). Параллельные версии прошли путь от крупнозернистых методов на распределенных системах [8, 9] до высокоэффективных мелкозернистых GPU-реализаций с использованием CUDA: GMMAS [10, 11], ACS-GPU-Alt [12], Multi-Colony ACO [13], параллельный MMAS [14] и реализация на основе warp specialization [15]. Во всех этих работах решение представляет собой гамильтонов цикл, а процесс его построения каждым агентом носит принципиально последовательный характер из-за необходимости отслеживания уже посещенных вершин (tabu-лист).

Данная работа выполнена в рамках государственного задания Минобрнауки России (номер темы FSFF-2026-0008) и посвящена разработке и анализу матричной модификации АСО для принципиально иного класса задач — задач параметрической оптимизации [16]. Здесь решение X представляет собой вектор значений n независимых параметров, выбор каждого из которых не зависит от других. Это фундаментальное структурное отличие открывает возможность для перехода от классической графовой парадигмы к формализму матричных операций и, как следствие, для создания качественно новых, более масштабируемых параллельных реализаций. В отличие от TSP, где параллелизм ограничен уровнем агентов, параметрическая задача допускает “поэлементный” параллелизм, когда потоки могут независимо обрабатывать отдельные параметры (слои графа). Это уникальное преимущество параметрической постановки было впервые использовано в работах [17, 18], однако в них основное внимание уделялось GPU-реализациям, тогда как систематический анализ параллельной сложности и эффективности на многоядерных CPU с OpenMP не проводился.

Целью настоящей работы является построение строгой теоретической базы для анализа эффективности матричного АСО и представление технологии конструирования параллельной программы для систем с общей памятью на основе OpenMP.

Дальнейшее изложение структурировано следующим образом. В разделе 2 вводится матричная формализация алгоритма и приводится анализ его теоретической вычислительной сложности. Раздел 3 посвящен теоретическому анализу параллельной сложности: строится информационный граф, вводится ярусно-параллельная форма, а также исследуются требования к памяти и вычислительная мощность алгоритма. Практические аспекты параллельной реализации с использованием технологии OpenMP, включая стратегии оптимизации и модель иерархической кэш-резидентности, детально рассматриваются в разделе 4. Экспериментальная верификация теоретических положений, сравнительный анализ масштабируемости на различных аппаратных архитектурах и блочная модификация алгоритма, направленная на преодоление ограничений памяти, представлены в разделе 5. В заключении сформулированы основные выводы и результаты работы.

2. Формализация матричного алгоритма и анализ вычислительной сложности.

2.1. Постановка параметрической задачи. В формализованном виде решение параметрической задачи может быть определено следующим образом [16]. Пусть имеется дискретное множество параметров $P = \{(p_1, p_2, \dots, p_i, \dots, p_n)\}$. У каждого i -го параметра существует множество допустимых значений $V_i = \{v_{1,i}, v_{2,i}, \dots, v_{j,i}, \dots, v_{m_i,i}\}$, $i = \overline{1, n}$. Количество допустимых значений определяется величиной $m_i > 0$, зависящей от номера параметра. В результате работы алгоритма оптимизации определяется вектор значений параметров (решение):

$$X_k = (x_{1,k}, x_{2,k}, \dots, x_{i,k}, \dots, x_{n,k}),$$

где $\forall i \exists j : (x_{i,k} = v_{j,i}) \wedge (v_{j,i} \in V_i)$. Найденный вектор отправляется на вычислитель (аналитическую или имитационную модель), который возвращает значение целевой функции $f(X_k)$. Решения рассматриваются в дискретном пространстве значений параметров, требований непрерывности и дифференцируемости на $f(X_k)$ не накладывается. Требуется определить оптимальное решение X_i^* [17].

Ключевым отличием параметрической задачи от классических применений АСО является независимость выбора значений параметров. Если в TSP решение — это упорядоченная последовательность вершин, то в параметрической задаче каждое значение x_i выбирается из своего множества V_i независимо от других. Для работы с непрерывными параметрами применяется процедура декомпозиции на слои, содержащие отдельные компоненты (например, десятичные разряды), а итоговое значение параметра определяется путем линейной свертки [18].

2.2. Матричная формализация алгоритма. В основе матричной модификации лежит представление всей информации, необходимой для работы алгоритма, в виде набора плотных матриц фиксированной размерности $n \times m$, где n — общее число слоев (параметров или компонент, полученных в результате

декомпозиции), а $m = \max_i m_i$ — максимальное число дискретных значений (вершин) в одном слое. Слой параметрического графа называется набор значений, из которого муравей-агент выбирает одно (например, отдельный параметр или отдельный компонент декомпозиции параметра). В работе [18] предложена структура параметрического графа в виде списка слоев и исследованы различные способы декомпозиции параметра на подслои. Показано, что оптимальная декомпозиция достигается путем разложения общего количества вершин для параметра на простые сомножители. Применительно к непрерывным параметрам сначала выполняется декомпозиция по разрядам (целые, десятичные, сотые и т.д.), после чего каждый разряд, содержащий 10 значений (цифр от 0 до 9), дополнительно разбивается на простые сомножители: $10 = 2 \times 5$. В результате максимальное число вершин в любом подслое не превышает 5, что определяет выбор $m = 5$ как максимального числа дискретных значений в одном слое параметрического графа.

Ключевыми матрицами, полностью описывающими состояние всей системы на итерации t , являются:

- $V \in \mathbb{R}^{n \times m}$ — статическая матрица значений параметров, содержащая конкретные числовые значения, соответствующие каждой вершине графа. Матрица дополняется фиктивными вершинами для унификации размерности и не изменяется в процессе работы алгоритма.
- $T(t) \in \mathbb{R}^{n \times m}$ — матрица феромонов, элемент $\tau_{i,j}(t)$ которой отражает коллективную предпочтительность выбора j -го значения для i -го параметра. Начальное значение обычно равно 1, все фиктивные вершины имеют нулевой феромон.
- $\Theta(t) \in \mathbb{N}^{n \times m}$ — матрица посещений, элемент $\theta_{i,j}(t)$ которой хранит суммарное количество случаев выбора агентами j -го значения для i -го параметра за всю историю поиска. Данная матрица является ключевым компонентом вероятностной формулы, решающей проблему преждевременной стагнации.

Условие канонического упорядочивания вершин. Пусть для параметра $i \in \{1, \dots, n\}$ множество допустимых значений имеет мощность $m_i < m$. Тогда выполняется следующее правило расположения: индексы $\forall j \in \{1, \dots, m_i\}$ соответствуют значимым вершинам, индексы $\forall j \in \{m_{i+1}, \dots, m\}$ — фиктивным. Начальное условие:

$$\tau_{i,j}(0) = 0, \quad \theta_{i,j}(0) = 0 : \forall i \in \{1, \dots, n\}, \quad \forall j \in \{m_{i+1}, \dots, m\}.$$

Следствием данного условия является сохранение нулевой привлекательности фиктивных вершин на любой итерации t , что гарантирует корректность алгоритма при матричной унификации размерности.

Итерационный цикл. С использованием введенных матриц один шаг алгоритма может быть записан в виде последовательности следующих матричных операторов.

Этап 1. Вычисление матрицы $F(t)$.

Нормализация феромонов:

$$T^{\text{norm}} = T(t) \oslash [T(t) \cdot I_{m \times m}],$$

где $I_{m \times m}$ — квадратная матрица размерности m , состоящая из единиц, а $\oslash$ — оператор поэлементного деления матриц, т.е. $\tau_{i,j}^{\text{norm}}(t) = \tau_{i,j} / \sum_{l=1}^m \tau_{i,l}$.

Вычисление матрицы привлекательности через аддитивную свертку:

$$Z(t) = \lambda_1 T^{\text{norm}}(t) + \lambda_2 \Theta^{o(-1)}(t) + \lambda_3 (\Theta(t) \oslash \Theta^{\text{max}}),$$

где $\lambda_1, \lambda_2, \lambda_3 \in \mathbb{R}_{>0}$ — весовые коэффициенты, $\Theta^{o(-1)}(t)$ — матрица, полученная поэлементным обращением $\Theta(t)$, а Θ^{max} — статическая матрица теоретически максимального числа посещений. Покоординатная запись для каждого элемента $z_{i,j}(t)$:

$$z_{i,j}(t) = \lambda_1 \tau_{i,j}^{\text{norm}}(t) + \frac{\lambda_2}{\theta_{i,j}(t)} + \lambda_3 \frac{\theta_{i,j}(t)}{\theta_{i,j}^{\text{max}}(t)}.$$

Итоговая матрица вероятностей $P(t)$ и матрица функций распределения $F(t)$:

$$P(t) = Z(t) \oslash [Z(t) \cdot I_{m \times m}], \quad F(t) = P(t) \cdot U_m,$$

где U_m — верхняя треугольная матрица размерности m , состоящая из единиц (оператор кумулятивной суммы). В покоординатной записи элементы матриц $P(t)$ и $F(t)$ соответственно равны:

$$p_{i,j}(t) = \frac{z_{i,j}(t)}{\sum_{k=1}^m z_{i,k}(t)}, \quad F_{i,j}(t) = \sum_{k=1}^j p_{i,k}(t).$$



Этап 2. Стохастическая генерация решений.

Для популяции из K агентов генерируется вспомогательная матрица случайных чисел $R(t) \in [0, 1)^{K \times n}$, элементы которой независимы и равномерно распределены. Для каждого агента $k \in \{1, \dots, K\}$ и каждого параметра $i \in \{1, \dots, n\}$ определяется индекс $s_{k,i}(t)$ выбранной вершины методом обратной функции:

$$s_{k,i}(t) = \min_{j \in \{1, \dots, m\}} \{r_{k,i}(t) \leq F_{i,j}(t)\}.$$

Совокупность всех индексов образует матрицу индексов $S(t) \in \mathbb{N}^{K \times n}$. Поскольку для последней значимой вершины $F_{i,m_i}(t) = 1$, условие всегда дает индекс $s_{k,i}(t) \leq m_i$, гарантируя выбор только допустимых значений.

На основе матрицы индексов $S(t)$ и статической матрицы значений V формируется матрица решений $X(t) \in \mathbb{R}^{K \times n}$:

$$x_{k,i}(t) = v_{i,s_{k,i}(t)}.$$

После построения всех решений производится их оценка целевой функцией f , формирующая вектор $Y(t) \in \mathbb{R}^K$:

$$y_k(t) = f(X_k(t)), \quad k = 1, \dots, K.$$

Этап 3. Обновление состояния колонии.

Пусть сформирована индикаторная матрица $I(t) \in \mathbb{N}^{n \times m}$, где элемент $I_{i,j}(t) = \sum_{k=1}^K [s_{k,i}(t) = j]$ равен числу агентов, выбравших j -е значение для i -го параметра. Тогда обновление матриц состояния записывается следующим образом:

$$T(t+1) = \rho T(t) + \Delta T(t), \quad \Theta(t+1) = \Theta(t) + I(t),$$

где ρ — коэффициент испарения феромона, а $\Delta T(t)$ — матрица вкладов феромона, вычисляемая на основе вектора $Y(t)$ и матрицы $I(t)$ (для минимизации целевой функции $\Delta \tau_{i,j}(t) = \sum_{k=1}^K I(t) \frac{1}{y_k(t)}$ при $Q = 1$, Q — параметр ACO¹).

2.3. Анализ теоретической вычислительной сложности. Под вычислительной сложностью в данной работе понимается асимптотическая оценка числа операций в RAM-модели (Random Access Machine), где все базовые арифметические операции (+, −, *, /), операции сравнения и логические операции считаются выполняемыми за одну условную единицу времени, т.е. имеют стоимость $O(1)$. Такой подход является стандартным в теоретическом анализе алгоритмов и позволяет сравнивать их масштабируемость по размерности задачи независимо от аппаратной платформы. Важно отметить, что в данной модели не учитываются эффекты иерархии памяти, которые будут рассмотрены отдельно в разделах, посвященных параллельной реализации и экспериментальной верификации.

Матричная формализация позволяет провести строгий теоретический анализ вычислительной сложности. Оценим временные затраты на выполнение одной итерации в указанной RAM-модели.

1. Нормализация феромонов. Для каждой из n строк матрицы $T(t)$ требуется вычислить сумму m элементов и выполнить m операций деления. Сложность — $O(n \cdot m)$.

2. Вычисление матрицы привлекательности. Для каждого из $n \cdot m$ элементов выполняется фиксированное число арифметических операций. Сложность — $O(n \cdot m)$.

3. Вычисление вероятностей и кумулятивных сумм. Для каждой из n строк выполняется суммирование m элементов, деление m элементов на сумму и последовательное суммирование m вероятностей. Сложность — $O(n \cdot m)$.

4. Генерация решений популяцией агентов. Для каждого из K агентов и каждого из n параметров выполняется поиск индекса $s_{k,i}(t)$ в строке матрицы $F(t)$, содержащей m элементов. В худшем случае может потребоваться проверка всех m элементов. Сложность — $O(K \cdot n \cdot m)$.

5. Обновление феромонов и матрицы посещений. Умножение всех $n \cdot m$ элементов на ρ — $O(n \cdot m)$. Добавление феромона и обновление посещений для каждого из K агентов и каждого из n параметров — $O(K \cdot n)$. Общая сложность — $O(n \cdot m + K \cdot n)$.

¹Подробнее см. Е. А. Davydkina, V. A. Nesterov, V. A. Sudakov et al., “Modifications of the ant Colonies Method for Solving a Discrete-Valued Parametric Problem,” J. Comput. Syst. Sci. Int. 64 (1), 46–62 (2025). doi 10.1134/S1064230725700042.

Суммируя сложности всех пяти этапов, получаем теоретическую оценку времени выполнения одной итерации матричного алгоритма:

$$O(n \cdot m) + O(n \cdot m) + O(n \cdot m) + O(K \cdot n \cdot m) + O(n \cdot m + K \cdot n).$$

Учитывая, что $K \gg 1$ и $n \gg 1$, а m — малая константа ($m = 5$ для графа оптимальной структуры), доминирующим членом является сложность генерации решений. Асимптотическая сложность одной итерации составляет:

$$O(K \cdot n \cdot m) \approx O(K \cdot n).$$

Полученная оценка демонстрирует фундаментальное преимущество матричной модификации для параметрических задач. В каноническом АСО для TSP сложность одной итерации составляет $O(K \cdot n^2)$, где n — число городов. Квадратичная зависимость от n возникает из-за того, что агент на каждом из n шагов выбирает из $O(n)$ доступных вершин с учетом *tabu*-листа. В матричной модификации, благодаря независимости выбора значений параметров, агент на каждом из n шагов выбирает из фиксированного и малого числа вариантов m . Поскольку $m \ll n$, сложность алгоритма становится линейной по числу параметров n . Этот переход от квадратичной зависимости к линейной является ключевым теоретическим результатом, который обосновывает высокую масштабируемость предложенного метода и его применимость для решения задач параметрической оптимизации большой размерности.

Важно подчеркнуть, что асимптотическая оценка сложности $O(K \cdot n)$ характеризует число операций алгоритма как функцию от размерности задачи. Переход к времени выполнения на реальном оборудовании требует учета аппаратных факторов: тактовой частоты процессора, эффективности использования конвейера, иерархии памяти и т.д. Поэтому в экспериментальной части работы измеряется непосредственно время выполнения, а теоретическая оценка сложности служит для предсказания масштабируемости алгоритма.

3. Теоретический анализ параллельной сложности.

3.1. Информационный граф и ярусно-параллельная форма. Строгая матричная формализация алгоритма позволяет построить точный информационный граф и его ярусно-параллельную форму (ЯПФ) [19, 20]. Для получения теоретически минимального времени выполнения при неограниченном параллелизме далее рассматривается каноническая ЯПФ — разбиение графа алгоритма на минимально возможное число ярусов. Обозначим через H высоту этой канонической ЯПФ, а через W — ее ширину (максимальное количество операций, которые могут быть выполнены одновременно на одном ярусе).

Этап 1. Вычисление матрицы $F(t)$.

Данный этап объединяет последовательность операций по преобразованию матриц состояния в матрицу функций распределения. Ключевым свойством является полная независимость по индексу i , т.е. операции с каждым слоем параметрического графа выполняются параллельно. В данный этап входят макрооперации:

1. Вычисление кумулятивной суммы столбца матрицы $T(t)$ за $m - 1 = 4$ операций сложения (при параллельной редукции — за $\lceil \log_2 m \rceil = 3$ операции).
2. Нормализация феромонов (поэлементное деление строк матрицы на сумму их элементов) за $m = 5$ операций сложения и $m = 5$ операций деления.
3. Аддитивная свертка для вычисления $Z(t)$: $2m = 10$ операций деления, $3m = 15$ операций умножения и $2m = 10$ операций сложения.
4. Построчная нормировка для получения $P(t)$ ($m = 5$ операций деления).
5. Построчное вычисление кумулятивных сумм для $F(t)$ ($m - 1 = 4$ операций сложения).

В итоге на этапе 1 для каждого слоя параметрического графа выполняется 58 операций (23 сложения, 20 делений, 15 умножений). В целом данный этап имеет ЯПФ с высотой H_1 и шириной W_1 :

$$H_1 = 11m - 2 \stackrel{m=5}{=} 53, \quad W_1 = n.$$

Этап 2. Генерация решений и оценка $Y(t)$.

Этап обладает полной независимостью как по муравьям-агентам K , так и по слоям (параметрам) n . Для каждого агента и каждого параметра выполняется:

1. Генерация случайного числа $r_{k,i}(t)$ (1 вызов генератора псевдослучайных чисел, имеющий фиксированную стоимость, не зависящую от n).



2. Циклический поиск позиции $s_{k,i}(t)$, имеющий неопределенное число итераций. В среднем, при равномерном распределении феромона, 2–3 операции сравнения.
3. Запись элемента в матрицу решений и проверка наличия в хэш-таблице [21].

Если параллелизм осуществляется только по муравьям-агентам, то высота и ширина ЯПФ второго этапа равны соответственно:

$$H_2 = n(1 + 2\log_2 m) + t_{\text{hash}} + t_{\text{search}} + t_{\text{add}}, \quad W_2 = K.$$

Если параллелизм осуществляется и по муравьям-агентам, и по отдельным слоям, то:

$$H'_2 = 1 + 2\log_2 m + t'_{\text{search}}, \quad W'_2 = nK.$$

Этап 3. Обновление матриц состояния.

Данный этап распадается на два принципиально различных по характеру параллелизма подэтапа.

1. *Испарение феромона*: умножение всей матрицы $T(t)$ на константу ρ , идеальный параллелизм, $H_{3.1} = 1$, $W_{3.1} = n \cdot m$.
2. *Добавление феромона и учет посещений*: операция с нерегулярным доступом к памяти и наличием конфликтов при записи. При использовании атомарных операций $H_{3.2} = 2K$, $W_{3.2} = n$. При параллелизме по слоям и агентам $H'_{3.2} = 2$, $W'_{3.2} = nK$.

Итоговые характеристики ЯПФ при предельном (теоретически максимальном) параллелизме, когда параллелизуются и агенты, и слои, а накладные расходы на синхронизацию не учитываются:

$$H' = 11m + 2\log_2 m + t'_{\text{search}} + 1 \overset{m=5}{\approx} 60 + t'_{\text{search}}, \quad W' = nK.$$

Для практического случая, когда число слоев n значительно превосходит число агентов K (что характерно для задач большой размерности) и учитываются накладные расходы на доступ к хэш-таблице:

$$H = 11m - 2 + n(1 + 2\log_2 m) + t_{\text{hash}} + t_{\text{search}} + t_{\text{add}} + 2n \overset{m=5}{\approx} 53 + 7n + t_{\text{search}}.$$

3.2. Анализ объемов данных и требований к памяти. Для полного анализа эффективности использования памяти (memory efficiency) необходимо оценить объемы данных, обрабатываемых на одной итерации. Под эффективностью использования памяти здесь понимается отношение объема полезных вычислений к объему данных, передаваемых между уровнями иерархии памяти, что непосредственно влияет на производительность memory-bound алгоритмов. При размерах типов $\text{sizeof}(\text{double}) = 8$ байт, $\text{sizeof}(\text{int}) = 4$ байта:

1. Данные на начало итерации.
 - Объем матрицы феромонов — $\text{sizeof}(T(t)) = n \times m \times 8 = 40n$ байт.
 - Объем матрицы посещений — $\text{sizeof}(\Theta(t)) = n \times m \times 4 = 20n$ байт.
 - Объем статической (read-only) матрицы параметров — $\text{sizeof}(V) = n \times m \times 8 = 40n$ байт.

Суммарный объем входных данных составляет $100n$ байт.

2. Промежуточные данные.

Матрицы $T^{\text{norm}}(t)$, $Z(t)$, $P(t)$ и $F(t)$ по $40n$ байт, а также дополнительные векторы кумулятивных сумм по $8n$ байт максимально занимают объем $176n$ байт. Если в программной реализации часть матриц может быть представлена векторами или скалярами, то общий объем промежуточных данных уменьшается до $40n + 56w$ байт, где w — число параллельных вычислительных контекстов (потоков/блоков).

3. Данные агентов.

Максимально данные агентов занимают в памяти $K(20n + 8)$ байт, поскольку $\text{sizeof}(X(t)) = n \times K \times 8 = 8Kn$ байт, $\text{sizeof}(Y(t)) = K \times 8 = 8K$ байт, $\text{sizeof}(R(t)) = n \times K \times 8 = 8Kn$ байт, $\text{sizeof}(S(t)) = n \times K \times 4 = 4Kn$ байт. Однако в программной реализации они занимают $16Kn + 16w$ байт, а минимальный объем дополнительных данных равен $8Kn + 16w$ байт.

4. Приватные буферы потоков.

При использовании техники локального накопления каждый из w потоков использует приватные копии матриц $T'(t)$ и $\Theta'(t)$ общим объемом $60nw$ байт.

Общий объем данных, требуемых алгоритму за три этапа:

$$\text{TotalSize} = n(140 + 16K + 60w) + 72w \text{ байт.} \quad (1)$$

Из данного соотношения можно определить максимальное количество слоев параметрического графа при фиксированных K , w и объеме доступной памяти:

$$n_{\max} = \frac{\text{TotalSize} - 72w}{140 + 16K + 60w}.$$

3.3. Вычислительная мощность и класс алгоритма. Принадлежность алгоритма к классу compute-bound или memory-bound является определяющей для его эффективной реализации на параллельных архитектурах. Определим метрику арифметической интенсивности ν (характеристика программной реализации) как отношение числа арифметических операций к объему переданных данных в байтах за одну итерацию.

Суммарное число арифметических операций за итерацию:

$$N_{\text{ops}} = n(18t_+ + 15t_+ + 10t_*) + K(n(t_{\text{rnd}} + 2t_+ + 2t_+) + t_{\text{hash}} + t_- + t_{\text{search}} + t_{\text{add}}) + 5nt_* + 2Knt_+.$$

Упрощенно (объединяя все типы операций):

$$N_{\text{ops}} \approx 48n + 7Kn + K(3 + t_{\text{search}}).$$

Суммарный объем данных, загружаемых и сохраняемых за итерацию, составляет TotalSize из формулы (1), для арифметической интенсивности получаем формулу

$$\nu = \frac{48n + 7Kn + K(3 + t_{\text{search}})}{140n + 16Kn + 60wn + 72w} \left[\frac{\text{оп}}{\text{байт}} \right].$$

В табл. 1 приведены значения арифметической интенсивности при варьировании количества агентов от 64 до 65536 и слоев параметрического графа от 42 до 1376256 при различном количестве вычислителей.

Результаты демонстрируют устойчивое снижение математического ожидания ν с 0.431 (при $w = 1$) до 0.320 (при $w = 64$), что составляет более 25%. Это является прямым следствием роста объема служебной памяти, выделяемой под приватные буферы потоков. Значения ν значительно ниже типичных показателей для compute-bound задач (например, более 10 оп/байт для перемножения плотных матриц). Это строго доказывает, что матричный АСО относится к классу memory-bound алгоритмов, и его производительность на реальном оборудовании в первую очередь ограничена пропускной способностью подсистемы памяти.

3.4. Характеристики эффективности параллельного алгоритма. На основе ЯПФ можно получить строгие оценки характеристик эффективности.

Предельное алгоритмическое ускорение при неограниченном параллелизме ($w \rightarrow \infty$) равно отношению последовательной сложности к высоте ЯПФ:

$$S_{\max} \approx \frac{O(K \cdot n \cdot m)}{O(n)} \approx K \cdot m = 5K.$$

Алгоритмическая эффективность определяется как отношение ускорения к ширине ЯПФ: $E = S/W \approx K/n$.

Таблица 1. Значения арифметической интенсивности ν (оп/байт) при варьировании K и n для различного числа потоков w

Table 1. Values of computational intensity ν (ops/byte) for varying K and n for different numbers of threads w

ν	w							
	1	2	4	8	12	16	32	64
$\nu_{\min}$	0.399	0.375	0.334	0.275	0.234	0.203	0.133	0.079
$\nu_{\max}$	0.443	0.443	0.443	0.443	0.443	0.443	0.443	0.442
ν_{average}	0.431	0.426	0.417	0.402	0.391	0.381	0.353	0.320

Алгоритмическая стоимость (сложность параллельного выполнения) определяется как произведение высоты ЯПФ на ширину: $C = H \cdot W \approx n(53 + 7n + t_{\text{search}})$.

Накладные расходы на параллелизацию оцениваются как доля дополнительных операций, возникающих из-за необходимости синхронизации и обмена данными между потоками. Основным источником накладных расходов является Этап 3, где требуется редукция вкладов от всех агентов. При использовании техники локального накопления объем синхронизаций сокращается с $O(K \cdot n)$ до $O(w \cdot n \cdot m)$. Относительные накладные расходы γ можно оценить как отношение объема синхронизаций к общему объему работы:

$$\gamma \approx \frac{w \cdot n \cdot m}{K \cdot n \cdot m} = \frac{w}{K}.$$

При $w \ll K$ (число потоков значительно меньше числа агентов) накладные расходы пренебрежимо малы ($\gamma \rightarrow 0$). При $w \approx K$ (каждый поток обрабатывает одного агента) накладные расходы становятся существенными ($\gamma \approx 1$).

4. Разработка параллельной программы на OpenMP. Теоретические выводы раздела 3 легли в основу методологии конструирования параллельной реализации для систем с общей памятью (SMP) с использованием OpenMP.

4.1. Детерминированные и стохастические участки алгоритма. В теории параллельного программирования детерминированным называется алгоритм, в котором отсутствуют условные переходы, зависящие от входных данных. Порядок выполнения операций в таком алгоритме полностью предопределен и не меняется в зависимости от конкретных значений обрабатываемых данных [20]. Рассмотрим с этой точки зрения этапы матричного АСО.

Этап 1 (вычисление матрицы $F(t)$) и этап 3 (обновление матриц) являются полностью детерминированными. Все циклы представляют собой циклы с параметром, в которых: число итераций строго фиксировано и равно либо n , либо K ; на каждой итерации выполняется одна и та же последовательность операций; отсутствуют условные переходы, зависящие от данных (data-dependent branches). Это означает, что при распределении таких циклов между потоками получаемые потоками объемы работ строго одинаковы, что является наилучшим сценарием для параллельных вычислений.

Этап 2 (генерация решений) содержит стохастическую составляющую, которая является основным источником дисбаланса нагрузки. Цикл поиска позиции $s_{k,i}(t)$ выполняется до тех пор, пока случайное число не окажется меньше или равно кумулятивной вероятности. Число итераций зависит от текущего распределения вероятностей и значения случайной величины. Внутри одного потока разные агенты могут завершить этот цикл за разное число шагов. Кроме того, обращение к глобальной хэш-таблице также вносит недетерминизм: время поиска и вставки зависит от заполненности таблицы.

4.2. Стратегии оптимизации распараллеливания. На эффективность работы программы влияют следующие директивы компилятора и методы взаимодействия с общими данными:

1. Локальное накопление для устранения конфликтов на этапе 3.

Прямое использование атомарных операций или критических секций для обновления матриц $T(t)$ и $\Theta(t)$ привело бы к катастрофическому падению производительности из-за ложного разделения кэш-линий и сериализации доступа. Вместо этого применяется техника локального накопления: каждый поток выделяет приватные локальные массивы размером $n \times m$, работает полностью независимо на протяжении всего цикла по агентам и только после обработки всех назначенных агентов однократно входит в критическую секцию для слияния результатов. Это сокращает количество синхронизаций с $O(K \cdot n)$ до $O(w)$. Используется директива `nowait`, которая позволяет потокам, завершившим обработку своих агентов, немедленно приступить к слиянию результатов, не дожидаясь отстающих.

2. Оптимизация доступа к хэш-таблице.

Для минимизации синхронизации при работе с глобальной хэш-таблицей используется техника шардирования (sharding): таблица разбивается на несколько независимых сегментов (шардов), ключ определяет номер шарда. Потоки, обращающиеся к разным шардам, могут выполнять операции полностью параллельно, без атомарных операций.

3. Параллелизм по параметрам на этапах 1 и 3.

Поскольку каждая строка имеет фиксированную длину $m = 5$, статическое планирование обеспечивает минимальные накладные расходы и оптимальную локальность данных в кэше. Каждый поток обрабатывает непрерывный блок строк, что способствует эффективной предвыборке данных. Директива `#pragma unroll` позволяет “раскрыть” внутренние циклы по 5 итераций.

4. Оптимизация для компилятора.

При объявлении функций используется ключевое слово `__restrict`, гарантирующее компилятору, что данные не перекрываются в памяти. Все основные вычислительные функции объявлены с ключевым словом `noexcept`, что исключает инфраструктуру обработки исключений.

4.3. Предотвращение ложного разделения кэш-линий (False Sharing). Одной из наиболее серьезных проблем, способных свести на нет весь выигрыш от распараллеливания на SMP-системах, является эффект ложного разделения кэш-линий. В архитектурах x86-64 размер кэш-линии составляет 64 байта. Когда ядро процессора модифицирует любые данные в кэш-линии, вся линия помечается как “модифицированная” (Modified) в его локальном кэше и становится “недействительной” (Invalid) в кэшах всех остальных ядер. Если другое ядро попытается прочитать или записать данные в той же кэш-линии, произойдет кэш-промах и линия будет принудительно синхронизирована между ядрами через общую шину.

В контексте матричного алгоритма АСО данная проблема проявляется в двух критических точках:

1. Параллельное обновление матриц $T(t)$ и $\Theta(t)$ на этапе 3.

Элементы матрицы феромонов `pheromone[i * M + 0]` и `pheromone[i * M + 1]` (оба типа `double`, по 8 байт) с большой вероятностью попадают в одну 64-байтную кэш-линию. Прямое атомарное обновление этих элементов разными потоками порождает “пинг-понг” кэш-линий. Техника локального накопления решает эту проблему: каждый поток выделяет свои приватные копии матриц в куче, и менеджер памяти ОС гарантирует, что разные динамические блоки памяти размещаются на разных страницах.

2. Использование массивов $S(t)$ и $X(t)$ на этапе 2.

Размер строки данных для одного агента составляет $n \times 8$ байт для $X(t)$ и $n \times 4$ байта для $S(t)$. При распределении агентов между потоками непрерывными блоками границы блоков выровнены по целому числу строк. Для предотвращения ложного разделения оптимальным является размер строки, кратный размеру кэш-линии: $n' = 64/8 = 8$ для `double`. Эффективное количество потоков равно $w' = \lfloor n/8 \rfloor$.

4.4. Иерархическая модель кэш-резидентности. Поскольку алгоритм является `memory-bound`, его производительность решающим образом определяется тем, насколько успешно рабочие структуры данных размещаются в иерархии кэш-памяти процессора.

Кэш ядра процессора L1 применяется для хранения информации для вычислительного конвейера одного ядра и оперирует окном данных. Для всех рассматриваемых процессоров объем окна данных на любом этапе алгоритма полностью помещается в кэш L1 ядра (для этапа 1 объем данных составляет 172 байта, для этапов 2 и 3 — 84 байта и 68 байт соответственно).

Кэш L2 хранит не только текущие данные, но и результаты работы аппаратного предвыборщика. Кэш L2, аналогично кэшу L1, выделяется на отдельное ядро. Критическим условием для производительности является *L2-резидентность приватных буферов*. Поскольку на этапе 3 доступ к приватному буферу является нерегулярным (индекс определяется путем агента), способность аппаратного предвыборщика эффективно подгружать данные из L3/DRAM при промахе мимо L2 крайне низка. Поэтому для обеспечения пиковой производительности необходимо, чтобы весь приватный буфер потока помещался в его L2-кэш. Если это условие нарушается, поток переходит в `latency-bound` режим, простаивая в ожидании данных из DRAM. В табл. 2 приведены ограничения для различных этапов на количество слоев графа, размещаемых в кэш L2.

Уровень L3 (8–32 Мбайт) является критическим для производительности матричного АСО. В L3-кэше размещаются суммарные объемы данных этапов по всем потокам. Поскольку L3 является общим ресурсом, условия формулируются для суммарного объема по всем логическим процессорам. Этап 1 имеет ограничение на несколько порядков выше, чем этапы 2 и 3, и поэтому практически всегда резидентен в L3-кэше. Этап 2 вытесняется из L3 уже при значениях n , лежащих в диапазоне от 1300 до 5100. Это означает, что для большинства практических задач данные этапа 2 не помещаются в L3. Однако, как показано выше, последовательный доступ к данным агентов позволяет предвыборщику эффективно загружать их из DRAM, и деградация этапа 2 остается умеренной. Но приватные буферы, создающиеся для каждого ядра процессора, имеют случайный доступ к памяти и для эффективной работы алгоритма должны быть полностью размещены в L3-кэше. В табл. 3 приведены оценки максимального количества слоев n , при котором приватные буферы полностью помещаются в L3-кэш различных процессоров.



Таблица 2. Ограничение на количество слоев графа n , размещаемых в кэш L2 одного ядра

Table 2. Limit on the number of graph layers n that can be placed in the L2 cache of a single core

Этап Stage	Ограничение Restriction	sizeof (L2) = 256 Кбайт sizeof (L2) = 256 KBytes	sizeof (L2) = 1024 Кбайт sizeof (L2) = 1024 KBytes
Этап 1 (полный) 1st stage (full)	$n \leq w \cdot \text{sizeof}(\text{L2})/100$	10485 ($w = 4$)	167772 ($w = 16$)
Этап 2 (полный, $K = 512$) 2nd stage (full, $K = 512$)	$n \leq w \cdot \text{sizeof}(\text{L2})/6224$	168 ($w = 4$)	2695 ($w = 16$)
Этап 2 (окно данных) 2nd stage (window of data)	$n \leq (\text{sizeof}(\text{L2}) - 40)/12$	21842	87378
Этап 3 (приватный буфер) 3rd stage (private buffer)	$n \leq \text{sizeof}(\text{L2})/60$	4369	17476
Этап 3 (полный, $K = 512$) 3rd stage (full, $K = 512$)	$n \leq \text{sizeof}(\text{L2})/(60 + 2108/w)$	446 ($w = 4$)	5468 ($w = 16$)

Таблица 3. Оценка максимального числа слоев n , при котором приватные буферы ($60nw$) полностью помещаются в L3-кэш

Table 3. Estimation of the maximum number of layers n for which private buffers ($60nw$) completely fit into the L3 cache

Процессор CPU model	sizeof (L3), Мбайт sizeof (L3), MBytes	Число потоков w Number of threads w	$n_{\max}$ для L3-кэша $n_{\max}$ for L3 cache
Intel Core i5-8300H (8th Gen)	8	8	≤ 17476
Intel Core i5-9400F (9th Gen)	9	6	≤ 26214
Intel Core i5-10400F (10th Gen)	12	12	≤ 17476
Intel Core i5-12450H (12th Gen)	12	12	≤ 17476
AMD Ryzen 5 7500F (Zen 4)	32	12	≤ 46603
AMD Ryzen 9 7900 (Zen 4)	64	24	≤ 46603
Intel Xeon Gold 6130 (2 proc.)	44 (2×22)	32	≤ 24054

5. Результаты экспериментального исследования.

5.1. Методика проведения эксперимента. Для валидации предложенного метода и верификации теоретической модели было разработано программное обеспечение на языке C++17 [22]. Тестирование проводилось на многомерных тестовых функциях (Растригина, Розенброка, Акли, “Сферической”, Грин-ванка, Захарова, Шейфеля, Леви, Михалевича–Викинского, Шаффера и др. [23]), которые выступали в роли параметрических моделей. Данные функции обладают хорошо изученными свойствами (многоэкстремальность, овражность, плоские области), что позволяет объективно и воспроизводимо сравнивать алгоритмы. В отличие от реальных задач, где результат зависит от множества факторов, на тестовых функциях можно изолированно оценить эффективность алгоритма.

Компиляция осуществлялась с использованием инструментальной цепочки Clang/LLVM (Clang 21.1.4) со следующими флагами:

```
clang++ -target x86_64-pc-windows-msvc -std=c++17 -fopenmp=libomp
        -O3 -mavx2 -march=native -o <executable> <source.cpp>
```

Исследования проводились на процессорах:

- Intel Xeon Gold 6130 2.1 ГГц (Skylake Server, 2017), 16 ядер $\times$ 2 процессора = 32 ядра, 64 логических потока, 256 Гбайт RAM DDR4-2666 (6 каналов, 128 Гбайт/с), L3 22 Мбайт $\times$ 2 = 44 Мбайт, L2 1024 Кбайт на ядро.
- Intel Core i5-12450H 2.0 ГГц (Alder Lake, 2021), 4 P-core + 4 E-core = 8 ядер, 12 логических потоков, 16 Гбайт RAM DDR5-3200 (2 канала, 51.2 Гбайт/с), L3 12 Мбайт, L2 1.25 Мбайт на P-core, 512 Кбайт на E-core.

- Intel Core i5-10400F 2.9 ГГц (Comet Lake, 2020), 6 ядер, 12 логических потоков (HT), 16 Гбайт RAM DDR4-2600, L3 12 Мбайт, L2 256 Кбайт на ядро.
- Intel Core i5-9400F 2.9 ГГц (Coffee Lake Refresh, 2019), 6 ядер, 16 Гбайт RAM DDR4-2933, L3 9 Мбайт, L2 256 Кбайт на ядро.
- Intel Core i5-8300H 2.3 ГГц (Coffee Lake, 2018), 4 ядра, 8 логических потоков (HT), 8 Гбайт RAM DDR4-2666, L3 8 Мбайт, L2 256 Кбайт на ядро.
- AMD Ryzen 9 7900 3.7 ГГц (Zen 4, Raphael, 2023), 12 ядер (2 CCD), 24 логических потока (SMT), 32 Гбайт RAM DDR5-6000, L3 64 Мбайт (2×32 Мбайт), L2 1024 Кбайт на ядро.
- AMD Ryzen 5 7500F 3.7 ГГц (Zen 4, Raphael, 2023), 6 ядер (1 CCD), 12 логических потоков (SMT), 32 Гбайт RAM DDR5-6000, L3 32 Мбайт, L2 1024 Кбайт на ядро.

Для каждой размерности задачи n (числа слоев параметрического графа от 42 до 1.38×10^6) и каждого числа потоков $w \in \{1, 2, \dots, \max_threads\}$ выполнялось 50 прогонов алгоритма с размером популяции $K = 512$ агентов. При этом измерялось нормированное на один слой параметрического графа время $t_{total}^{norm}(n, w)$ в мс/слой. Результаты представляют собой оценку математического ожидания и доверительные интервалы (уровень доверия 0.99) по 50 прогонам после двух “холостых” итераций для разогрева кэша.

Для количественной оценки того, насколько полно параллельная программа использует вычислительные ресурсы многоядерной системы, в работе применяются стандартные метрики ускорения и эффективности.

Масштабируемость характеризует способность алгоритма увеличивать производительность пропорционально росту числа используемых потоков. Идеальная масштабируемость достигается, когда производительность возрастает в w раз при задействовании w потоков по сравнению с однопоточным выполнением.

Ускорение $S_w(n)$ показывает, во сколько раз быстрее выполняется задача при использовании w потоков по сравнению с однопоточным выполнением:

$$S_w(n) = \frac{t_{total}^{norm}(n, 1)}{t_{total}^{norm}(n, w)}.$$

Значение $S_w(n) = w$ соответствует идеальной линейной масштабируемости.

Эффективность $E_w(n)$ характеризует долю полезного использования каждого потока и вычисляется как нормированное ускорение:

$$E_w(n) = \frac{S_w(n)}{w}.$$

При идеальной масштабируемости $E_w(n) = 1$.

На практике из-за накладных расходов на синхронизацию, дисбаланса нагрузки, конкуренции за кэш-память и пропускную способность памяти обычно $E_w(n) < 1$. Снижение эффективности при росте w свидетельствует о том, что добавление новых потоков дает все меньший прирост производительности, что особенно характерно для memory-bound алгоритмов при исчерпании пропускной способности подсистемы памяти. В отдельных случаях может наблюдаться суперлинейное ускорение ($S_w(n) > w$, $E_w(n) > 1$). Такой эффект не противоречит теории параллельных вычислений и, как правило, объясняется тем, что с ростом числа потоков увеличивается суммарный объем кэш-памяти, доступной для обработки, благодаря чему больший объем рабочего набора размещается в быстрой памяти на кристалле и сокращается число обращений к оперативной памяти.

5.2. Анализ масштабируемости и верификация модели кэш-резидентности. На рис. 1 представлена зависимость нормированного времени выполнения различных этапов алгоритма от размерности параметрического графа для разных процессоров при максимальном количестве потоков w . Видно, что во всех конфигурациях в режиме нормального исполнения, когда данные матричного АСО для всех потоков могут быть размещены в кэше L3, наибольшее время занимает этап 2 (зеленая линия на рис. 1). Но после того, как суммарный объем локальных матриц ($60nw$) превысит объем кэша L3, существенное лидерство по времени исполнения занимает этап 3, на котором как раз и используются локальные матрицы для накопления изменения матриц феромонов и посещений.

В табл. 4 приведены полные результаты для процессора Intel Core i5-8300H, который является эталонным примером, позволяющим проследить все режимы работы алгоритма.

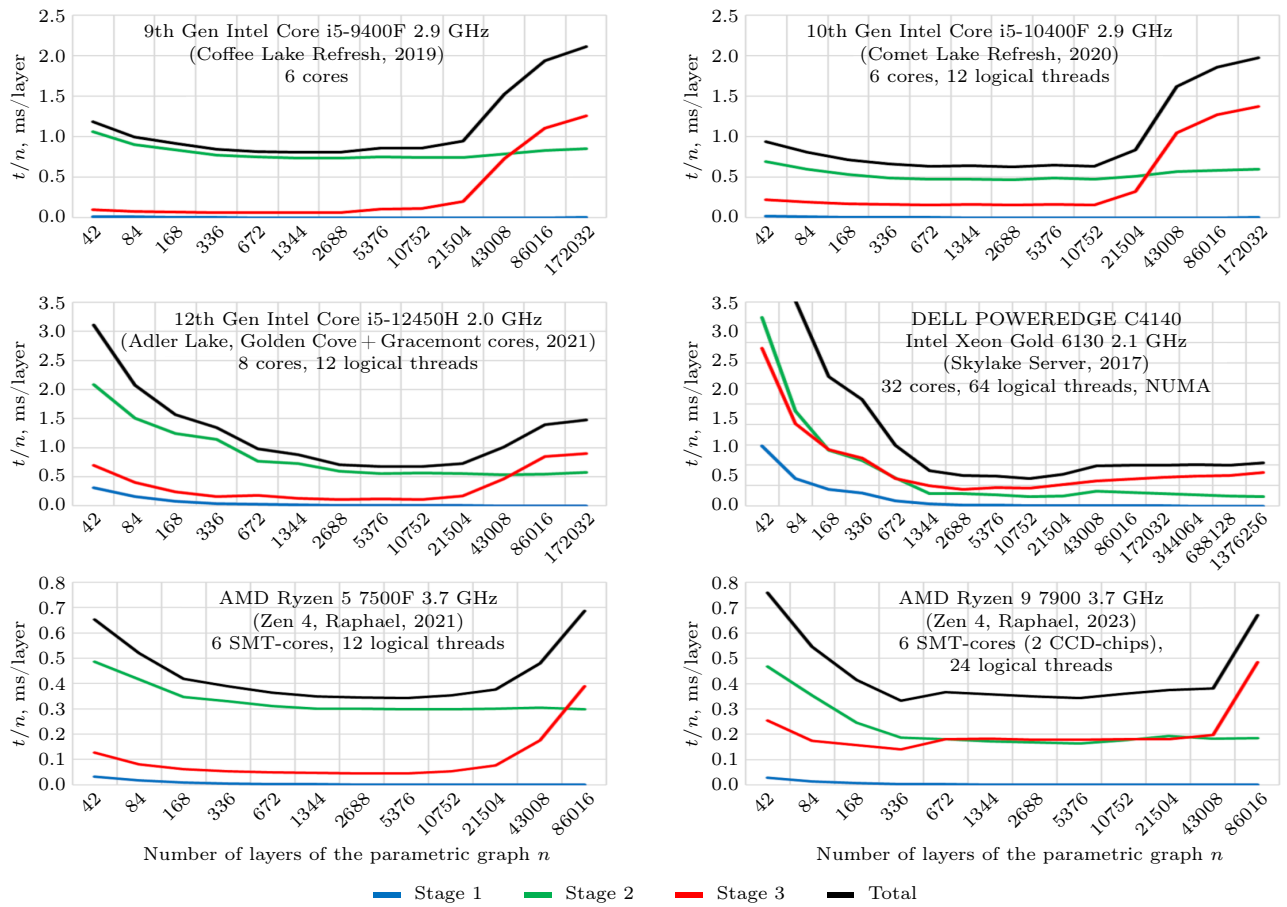


Рис. 1. Зависимость времени выполнения этапа алгоритма от размерности параметрического графа для различных процессоров, в мс/слой

Fig. 1. Dependence of the time for executing an algorithm stage on the dimension of the parametric graph for different processors, in ms/layer

Таблица 4. Оценки времени (мс/слой), ускорения и эффективности для Intel Core i5-8300H в зависимости от количества слоев n и числа потоков w

Table 4. Time (ms/layer), speedup, and efficiency for Intel Core i5-8300H depending on number of layers n and number of threads w

n	Параметр Parameter	w							
		1	2	3	4	5	6	7	8
42	t/n	5.77	3.01	2.14	1.73	1.60	1.39	1.28	1.19
	S_w	1.00	1.92	2.69	3.33	3.61	4.14	4.50	4.85
	E_w	1.00	0.96	0.90	0.83	0.72	0.69	0.64	0.61
2688	t/n	4.51	2.28	1.53	1.20	1.23	1.06	0.95	0.88
	S_w	1.00	1.98	2.94	3.77	3.68	4.26	4.74	5.15
	E_w	1.00	0.99	0.98	0.94	0.74	0.71	0.68	0.64
10752	t/n	4.66	2.36	1.59	1.28	1.31	1.13	1.02	0.97
	S_w	1.00	1.97	2.93	3.63	3.57	4.11	4.56	4.83
	E_w	1.00	0.99	0.98	0.91	0.71	0.69	0.65	0.60
43008	t/n	4.72	2.44	1.86	1.88	1.95	1.99	2.38	2.77
	S_w	1.00	1.93	2.54	2.52	2.42	2.38	1.98	1.70
	E_w	1.00	0.97	0.85	0.63	0.48	0.40	0.28	0.21
86016	t/n	4.87	3.25	2.96	2.94	2.92	2.93	3.04	3.09
	S_w	1.00	1.50	1.65	1.66	1.67	1.66	1.60	1.58
	E_w	1.00	0.75	0.55	0.41	0.33	0.28	0.23	0.20
172032	t/n	5.58	3.90	3.46	3.28	3.27	3.24	3.26	3.25
	S_w	1.00	1.43	1.61	1.70	1.71	1.72	1.71	1.72
	E_w	1.00	0.72	0.54	0.43	0.34	0.29	0.24	0.21

Анализ табл. 4 выявляет четыре характерных режима работы алгоритма:

1. Кэш-резидентность в L3 ($n \leq 10752$).

При $n \leq 10752$ объем приватных буферов одного потока составляет не более $60 \times 10752 \approx 645$ Кбайт, суммарный объем при $w = 8$ — около 5.16 Мбайт, что полностью помещается в L3-кэш (8 Мбайт). Эффективность при $w = 4$ (физические ядра) составляет $E_4 \in [0.83, 0.95]$, при $w = 8$ (логические потоки) $E_8 \in [0.61, 0.70]$. Ускорение S_8 может достигать значений в диапазоне от 4.83 до 5.61. Снижение эффективности при переходе от $w = 4$ к $w = 8$ объясняется включением логических ядер Hyper-Threading, которые разделяют ресурсы физических ядер. Средняя эффективность одного логического ядра η_{HT} составляет примерно 0.3...0.5 от эффективности физического.

2. Переходный ($n \approx 21504$).

При $n = 21504$ объем приватных буферов при $w = 8$ составляет $60 \times 21504 \times 8 \approx 10.3$ Мбайт, что начинает превышать объем L3-кэша (8 Мбайт). Эффективность падает до $E_8 = 0.44$, ускорение — до $S_8 = 3.49$.

3. Вытеснение из L3 ($n \approx 43008$).

При $n = 43008$ объем приватных буферов при $w = 8$ составляет ≈ 20.6 Мбайт, что в 2.5 раза превышает L3-кэш. Наблюдается катастрофическая деградация: $E_8 = 0.21$, $S_8 = 1.70$. Ускорение S_8 практически не отличается от $S_4 = 2.52$.

4. Полный latency-bound ($n \geq 86016$).

При $n \geq 86016$ ускорение S_8 при каждом прогоне независимо от числа потоков принимает значения из диапазона от 1.58 до 1.72, что можно считать стабильным показателем. Алгоритм полностью перешел в режим, в котором производительность определяется задержками DRAM.

5.3. Сравнительный анализ масштабируемости на различных процессорах. Сравнение процессоров Intel Core i5-8300H (4 ядра + HT) и Intel Core i5-9400F (6 ядер без HT) при $n \leq 10752$ показывает, что эффективность одного физического ядра Intel Core i5-9400F ($E_6 = 0.88$) практически совпадает с эффективностью одного физического ядра Intel Core i5-8300H ($E_4 = 0.91$). При $n = 43008$ (latency-bound режим) Intel Core i5-9400F обеспечивает 3.1-кратное ускорение, в то время как для Intel Core i5-8300H ускорение падает до 1.70. Более высокий порог деградации для Intel Core i5-9400F объясняется как большим объемом L3 (9 Мбайт против 8 Мбайт), так и отсутствием Hyper-Threading, снижающего конкуренцию за кэш.

Intel Core i5-10400F (6 ядер + HT) при $n = 10752$ показывает ускорение вычислений $S_{12} = 7.10$, что лучше, чем $S_6 = 5.29$ для Intel Core i5-9400F. Эффективность Hyper-Threading составляет $\eta_{\text{HT}} \approx 0.35$. При $n = 43008$ ускорение S_{12} падает до 3.10, а при $n = 86016$ — до 2.64.

В табл. 5 приведены результаты для гибридного процессора Intel Core i5-12450H.

Анализ табл. 5 позволяет выделить три характерных паттерна прироста ускорения, соответствующих различным типам задействуемых ядер:

1. Паттерн P-core.

Ускорение с одного до четырех потоков при $n \geq 672$ демонстрирует стабильно высокий прирост ΔS от 0.8 до 1.0 на каждый добавляемый поток. Это соответствует добавлению физического P-core (архитектура Golden Cove) с L2-кэшем 1.25 Мбайт.

2. Паттерн включения Hyper-Threading после P-core ($\Delta S < 0$).

Переход от четырех потоков к пяти при $n \leq 10752$ демонстрирует отрицательный прирост ΔS от -0.09 до -0.49 . Это однозначно указывает на добавление Hyper-Threading-потока на P-core, который создает конкуренцию за L2-кэш и исполнительные порты.

3. Паттерн E-core.

Переходы после пяти потоков при $n \geq 336$ демонстрируют небольшой, но стабильный положительный прирост ΔS от 0.3 до 0.4. Это соответствует добавлению E-core (архитектура Gracemont), которое дает в 2–4 раза меньший прирост, чем P-core.

При использовании только P-ядер ($w = 4$) доля барьерной синхронизации составляет 12.5% времени выполнения. Добавление Hyper-Threading-потока на P-ядро ($w = 5$) приводит к росту времени ожидания на барьерах до 18.2%, что объясняет отрицательный прирост ускорения ($\Delta S \approx -0.09$). При использовании E-ядер ($w = 6$) доля синхронизации снижается до 15.8%, однако производительность каждого E-ядра в



Таблица 5. Оценки времени (мс/слой), ускорения и эффективности для Intel Core i5-12450H (12-е поколение, P/E-ядра) без использования привязки потоков к ядрам

Table 5. Time (ms/layer), speedup, and efficiency for Intel Core i5-12450H (12th Gen, P/E-cores) without thread affinity

n	Параметр Parameter	w											
		1	2	3	4	5	6	7	8	9	10	11	12
42	t/n	5.09	3.37	3.03	3.31	3.17	3.07	3.09	3.01	3.10	3.28	3.29	3.11
	S	1.00	1.51	1.68	1.54	1.61	1.66	1.65	1.69	1.64	1.55	1.55	1.64
	E	1.00	0.75	0.56	0.38	0.32	0.28	0.24	0.21	0.18	0.16	0.14	0.14
2688	t/n	3.88	2.01	1.44	1.11	1.21	1.12	0.99	0.92	0.94	0.87	0.78	0.71
	S	1.00	1.93	2.69	3.51	3.22	3.47	3.91	4.20	4.15	4.47	5.00	5.47
	E	1.00	0.96	0.90	0.88	0.64	0.58	0.56	0.53	0.46	0.45	0.45	0.46
10752	t/n	3.85	2.02	1.40	1.09	1.11	1.02	0.93	0.84	0.79	0.74	0.70	0.68
	S	1.00	1.91	2.75	3.55	3.46	3.76	4.15	4.58	4.88	5.22	5.51	5.71
	E	1.00	0.95	0.92	0.89	0.69	0.63	0.59	0.57	0.54	0.52	0.50	0.48
21504	t/n	3.91	2.22	1.44	1.15	1.13	1.05	0.96	0.88	0.83	0.78	0.75	0.73
	S	1.00	1.76	2.72	3.41	3.45	3.72	4.08	4.45	4.74	5.02	5.25	5.38
	E	1.00	0.88	0.91	0.85	0.69	0.62	0.58	0.56	0.53	0.50	0.48	0.45
43008	t/n	4.76	2.71	1.81	1.25	1.26	1.15	1.07	1.00	0.95	0.93	0.95	1.01
	S	1.00	1.76	2.64	3.80	3.77	4.13	4.43	4.77	5.02	5.12	5.03	4.73
	E	1.00	0.88	0.88	0.95	0.75	0.69	0.63	0.60	0.56	0.51	0.46	0.39
86016	t/n	4.77	2.35	1.69	1.45	1.45	1.44	1.44	1.43	1.41	1.41	1.41	1.39
	S	1.00	2.00	2.81	3.28	3.28	3.31	3.30	3.32	3.37	3.39	3.38	3.42
	E	1.00	1.00	0.94	0.82	0.66	0.55	0.47	0.42	0.37	0.34	0.31	0.28

2–4 раза ниже производительности P-ядра, что соответствует приросту $\Delta S \approx 0.3 \dots 0.4$. При $n = 86016$ все ΔS для $w \geq 5$ близки к нулю, что соответствует latency-bound режиму.

Для подтверждения предложенной модели кэш-резидентности проведен анализ аппаратных счетчиков производительности с использованием Intel VTune на процессоре Intel Core i5-12450H (Alder Lake) для всех четырех режимов масштабируемости.

При переходе от $n = 10752$ к $n = 86016$ количество промахов L3 (LLC Miss Count) возрастает в 43 раза, что является прямым доказательством вытеснения приватных буферов из кэш-памяти последнего уровня (табл. 6). Доля циклов, затрачиваемых на ожидание данных из DRAM (DRAM Bound), растет с 1.4% до 49.6%, среднее число тактов на инструкцию (CPI) возрастает с 1.167 до 2.848, а доля полезной работы (Retiring) падает с 16.4% до 7.6%. Эти данные позволяют определить падение ускорения с 5.6 до 1.6 раз при переходе от режима кэш-резидентности к latency-bound режиму.

Измерения аппаратных счетчиков показывают, что характер роста числа промахов L3 при увеличении числа потоков зависит от режима. В режиме кэш-резидентности ($n = 10752$) метрика LLC Miss Count растет медленно: с 80 млн ($w = 1$) до 154 млн ($w = 12$). В режиме вытеснения из L3 ($n = 43008$) наблюдается значительно более резкий рост метрики: с 1.16 млрд до 4.61 млрд, причем основной скачок происходит при переходе от $w = 4$ к $w = 8$ (с 1.39 млрд до 4.05 млрд). Это указывает на пороговый эффект: при малом числе потоков данные еще удерживаются в кэше, а при превышении критического числа потоков происходит массовое вытеснение. Также важно, что в режиме кэш-резидентности ($n = 10752$) CPI растет плавно с 0.905 до 1.167, а доля полезной работы Retiring достигает максимума при $w = 4$ и остается

Таблица 6. Ключевые аппаратные метрики для различных режимов масштабируемости на Intel Core i5-12450H, $w = 12$

Table 6. Key hardware metrics for different scalability modes on Intel Core i5-12450H, $w = 12$

Метрика Metric	n			
	10752	21504	43008	86016
LLC Miss Count	1.54×10^8	2.36×10^9	4.61×10^9	6.66×10^9
DRAM Bound (% Clockticks)	1.4%	6.7%	32.9%	49.6%
CPI	1.167	1.299	1.944	2.848
Retiring (% Pipeline Slots)	16.4%	15.0%	11.0%	7.6%

стабильной. В переходном режиме ($n = 21504$) наблюдается ускоренный рост CPI при $w > 8$, а DRAM Bound возрастает до 6.7%. В режиме вытеснения из L3 ($n = 43008$) DRAM Bound скачкообразно растёт при переходе от $w = 4$ к $w = 8$ (с 1.6% до 11.3%) и достигает 32.9% при $w = 12$. А в latency-bound режиме ($n = 86016$) DRAM Bound уже при $w = 4$ составляет 18.1%, а при $w = 12$ достигает 49.6%.

В табл. 7 приведено сравнение ускорения с явной привязкой потоков к ядрам (affinity) и без нее. Для эксперимента с привязкой потоков к ядрам на процессоре Intel Core i5-12450H использовалась стратегия последовательного закрепления потоков за ядрами в порядке их нумерации. В процессорах Intel Alder Lake ядра нумеруются следующим образом: сначала логические процессоры, соответствующие Р-ядрам (ядра 0–3), затем Е-ядра (ядра 4–7), и, наконец, Hyper-Threading-потоки на Р-ядрах (ядра 8–11).

При малых n конфигурация с affinity демонстрирует существенное преимущество (до 2.06 раз при $w = 4$ для $n = 42$), так как предотвращает преждевременное размещение потоков на Hyper-Threading-ресурсах. При $n = 10752$, когда приватные буферы не помещаются в L2-кэш Е-ядер (512 Кбайт), конфигурация с affinity проигрывает, поскольку принудительное закрепление потоков за Е-ядрами приводит к их latency-bound деградации.

Процессоры AMD на базе микроархитектуры Zen 4 демонстрируют масштабируемость, обусловленную большим объемом кэш-памяти L3. AMD Ryzen 9 7900 (12 ядер, 2 CCD, L3 64 Мбайт) показывает превосходную производительность: при $n = 10752$ ускорение достигает $S_{18} = 10.0$, а при $n = 43008$ — $S_{18} = 10.15$. Катастрофической деградации, характерной для процессоров Intel с малым объемом L3, не наблюдается вплоть до $n = 43008$, что подтверждает фундаментальную роль общего кэша L3 в масштабируемости memory-bound алгоритмов. Эффективность SMT для Zen 4 невысока ($\eta_{\text{SMT}} \approx 0.05 \dots 0.10$), что значительно ниже эффективности Hyper-Threading на Intel (0.30...0.50). Это объясняется тем, что универсальные SIMD-порты Zen 4 создают более острую конкуренцию между SMT-потоками за исполнительные ресурсы. Более доступный Ryzen 5 7500F (6 ядер, 1 CCD, L3 32 Мбайт) также показывает отличную эффективность, особенно на физических ядрах. При $n = 10752$ ускорение на 6 физических ядрах достигает $S_6 = 5.68$ (эффективность $E_6 = 0.95$), что является наивысшим показателем среди всех протестированных конфигураций и объясняется универсальностью SIMD-портов Zen 4 и отсутствием межкристалльных задержек Infinity Fabric.

Исследование на двухпроцессорной серверной платформе Intel Xeon Gold 6130 (32 физических ядра, 2 сокета, NUMA) выявило особенности масштабирования, характерные для NUMA-архитектур. Анализ данных табл. 8 показывает, что при переходе от 16 потоков (один сокет) к 32 потокам (два сокета) наблюдается замедление для всех размерностей: для $n = 42$ время возрастает с 3.14 до 4.07 мс/слой, для $n = 10752$ — с 0.57 до 0.60 мс/слой, для $n = 43008$ — с 0.57 до 0.67 мс/слой, для $n = 172032$ — с 0.92 до 0.86 мс/слой, для $n = 1376256$ — с 0.87 до 0.88 мс/слой. При использовании 64 логических потоков (Hyper-Threading на двух сокетах) наблюдается дальнейшая деградация производительности.

Вместе с тем для задач большой размерности ($n \geq 172032$) ускорение на 32 потоках остается на уровне $S_{32} \approx 8.5 \dots 9.9$, что лишь незначительно уступает пиковым значениям $S_{24} \approx 10.1 \dots 10.7$. Это свидетельствует о том, что при достаточном объеме вычислений алгоритм способен эффективно использовать вычислительные ресурсы обоих сокетов, несмотря на NUMA-накладные расходы. Однако для достижения максимальной эффективности на NUMA-системах необходима оптимизация размещения данных с использованием политики first-touch и явная привязка потоков к конкретным NUMA-узлам, что является предметом дальнейших исследований.

5.4. Сводная таблица производительности для размерности $n = 10752$. Для наглядного сравнения производительности на различных процессорных архитектурах в табл. 9 приведены значения нормированного времени (мс/слой) для размерности параметрического графа $n = 10752$ (режим устойчивой кэш-резидентности для большинства процессоров) и размера популяции $K = 512$ агентов.

Таблица 7. Отношение ускорения с привязкой потоков к ядрам к ускорению без привязки для Intel Core i5-12450H

Table 7. Ratio of speedup with affinity to speedup without affinity for Intel Core i5-12450H

n	w											
	1	2	3	4	5	6	7	8	9	10	11	12
42	0.92	1.17	1.42	2.06	1.77	1.94	2.17	2.35	2.59	1.55	1.70	1.60
672	0.96	1.01	1.05	1.22	1.16	1.13	1.16	1.09	1.09	0.61	0.66	0.65
10752	0.97	1.02	1.00	0.99	0.79	0.80	0.74	0.66	0.68	0.41	0.42	0.43



Таблица 8. Оценки времени (мс/слой), ускорения и эффективности для Intel Xeon Gold 6130 в зависимости от количества слоев n и числа потоков w

Table 8. Time (ms/layer), speedup, and efficiency for Intel Xeon Gold 6130 depending on number of layers n and number of threads w

n	Параметр Parameter	w							
		1	2	4	8	16	24	32	64
42	t/n	8.80	7.24	5.13	5.10	3.14	3.66	4.07	9.99
	S_w	1.00	1.21	1.71	1.72	2.80	2.40	2.16	0.88
	E_w	1.00	0.61	0.43	0.22	0.18	0.10	0.07	0.01
10752	t/n	5.83	3.20	1.61	0.88	0.57	0.51	0.60	0.68
	S_w	1.00	1.82	3.62	6.63	10.20	11.40	9.71	8.57
	E_w	1.00	0.91	0.91	0.83	0.64	0.48	0.30	0.13
43008	t/n	6.25	3.32	1.69	0.91	0.57	0.53	0.67	0.99
	S_w	1.00	1.88	3.70	6.87	11.00	11.80	9.33	6.31
	E_w	1.00	0.94	0.93	0.86	0.69	0.49	0.29	0.10
172032	t/n	8.51	5.08	2.47	1.34	0.92	0.84	0.86	1.00
	S_w	1.00	1.68	3.44	6.34	9.25	10.10	9.89	8.51
	E_w	1.00	0.84	0.86	0.79	0.58	0.42	0.31	0.13
1376256	t/n	8.65	3.94	2.06	1.21	0.87	0.81	0.88	1.05
	S_w	1.00	2.20	4.20	7.15	9.94	10.70	9.83	8.24
	E_w	1.00	1.10	1.05	0.89	0.62	0.45	0.31	0.13

Таблица 9. Сводная таблица времени выполнения (мс/слой) для $n = 10752$, $K = 512$ на различных процессорах

Table 9. Summary table of execution time (ms/layer) for $n = 10752$, $K = 512$ on different processors

Процессор, (ядра/потоки) CPU model, (cores/threads)	sizeof (L3), Мбайт sizeof (L3), Mbytes	w						
		1	2	4	6	8	12	16–24
Intel Core i5-8300H, (4/8)	8	4.66	2.36	1.28	1.13	0.97	—	—
Intel Core i5-9400F, (6/6)	9	4.56	2.39	1.25	0.86	0.84	—	—
Intel Core i5-10400F, (6/12)	12	3.91	2.01	1.05	0.74	0.69	0.55	—
Intel Core i5-12450H, (4P+4E/12)	12	3.85	2.02	1.09	1.02	0.84	0.68	—
AMD Ryzen 5 7500F, (6/12)	32	3.04	1.52	0.79	0.55	0.44	0.37	—
AMD Ryzen 9 7900, (12/24)	64	2.84	1.42	0.73	0.52	0.42	0.32	0.28 ($w = 18$)
Intel Xeon Gold 6130, (32/64)	44 (2×22)	5.83	3.20	1.61	1.09	0.88	0.51	0.57 ($w = 16$)

Анализ табл. 9 позволяет сделать следующие выводы:

- Лидером по абсолютной производительности является AMD Ryzen 9 7900, который достигает минимального времени 0.28 мс/слой на 18 потоках, что в 10.1 раза быстрее его однопоточного режима и в 3.4 раза быстрее, чем лучший результат среди потребительских процессоров Intel (Intel Core i5-10400F, 0.55 мс/слой).
- Лучшую эффективность на физических ядрах демонстрирует AMD Ryzen 5 7500F, который на 6 потоках показывает время 0.55 мс/слой при эффективности $E_6 = 0.95$. Такой показатель эффективности является наилучшим среди всех протестированных конфигураций.
- Применение технологии Hyper-Threading в процессоре Intel Core i5-10400F является оправданным решением (0.69 мс/слой на 8 потоках против 0.55 мс/слой на 12 потоках), в то время как технология SMT в процессоре AMD Ryzen 9 7900 показывает лишь незначительный выигрыш при вычислениях (снижение с 0.32 мс/слой на 12 потоках до 0.28 мс/слой на 18 потоках).
- Intel Xeon Gold 6130, несмотря на более высокое абсолютное время в однопоточном режиме (5.83 мс/слой из-за более низкой тактовой частоты), демонстрирует впечатляющее ускорение до 11.4 на 24 потоках (0.51 мс/слой), что подтверждает эффективность использования многих ядер для memory-bound алгоритмов и высокую масштабируемость алгоритма на серверной платформе.

5.5. Влияние размера популяции на масштабируемость. Для обоснования выбора размера популяции $K = 512$ проведено исследование влияния этого параметра на масштабируемость алгоритма. Эксперименты выполнялись на процессоре Intel Core i5-9400F (6 физических ядер) при использовании



мым совершая “пустые” вычисления, не вносящие вклад в исследование пространства. Аналогично потеря или искажение данных от одного или нескольких агентов из-за состояния гонки не является уникальным или критическим событием. Проведенный анализ точности найденного решения показал, что для задач большой размерности (более 200 слоев параметрического графа) состояние гонки не оказывает статистически значимого отклонения качества решения. Это позволяет рассматривать “алгоритм без блокировок” не просто как инструмент для измерения накладных расходов, а как допустимую с практической точки зрения оптимизацию для класса задач параметрической оптимизации большой размерности.

В табл. 11 приведены результаты сравнительного анализа алгоритма с локальным накоплением и алгоритма без блокировок, в котором потоки непосредственно обновляют глобальные матрицы без каких-либо критических секций или атомарных операций.

Значение > 1 означает преимущество алгоритма без блокировок, < 1 — преимущество алгоритма с локальным накоплением. При малых n алгоритм с локальным накоплением демонстрирует подавляющее преимущество (ускорение до 2.35 при $n = 42$, $w = 8$) благодаря устранению ложного разделения кэш-линий. При $n = 43008$ и особенно при $n = 86016$ наблюдается инверсия эффективности: алгоритм без блокировок начинает выигрывать (до 1.82 раз при $n = 86016$, $w = 12$). Причина заключается в том, что при больших n суммарный объем приватных буферов ($60nw$) многократно превышает объем L3-кэша, вызывая массовые обращения к DRAM, тогда как глобальные матрицы в алгоритме без блокировок ($60n$ байт суммарно) полностью помещаются в L3-кэш, и трафик когерентности обходится дешевле, чем промахи в L3 и обращения к DRAM.

5.7. Анализ эффективности использования подсистемы памяти. Сравним реально достигнутую производительность с теоретическим пределом, определяемым пропускной способностью и латентностью памяти, для двух принципиально различных режимов работы: режима L3-резидентности ($n \leq 10752$, все структуры данных размещены в L3-кэше) и режима вытеснения из L3 ($n \geq 43008$, приватные буферы потоков вытеснены в DRAM). Реальная средняя пропускная способность памяти определяется как $BW_{\text{real}} = V_{\text{total}}/T_{\text{total}}$, где V_{total} — суммарный объем данных, переданных через критический интерфейс памяти за 500 итераций, а T_{total} — общее время 500 итераций в секундах. Эффективность использования памяти оценивается как $\eta_{BW} = BW_{\text{real}}/BW_{\text{peak}}$, где BW_{peak} — реальная пиковая пропускная способность, измеренная бенчмарком STREAM (операция Triad) для каждого процессора в многопоточном режиме. Для режима L3-резидентности критическим интерфейсом является $L2 \leftrightarrow L3$, и в качестве BW_{peak} используется измеренная пропускная способность этого интерфейса, составляющая около 50 Гбайт/с для Intel Coffee Lake и до 90 Гбайт/с для AMD Zen 4. Для режима вытеснения из L3 критическим интерфейсом является $L3 \leftrightarrow \text{DRAM}$. В режиме L3-резидентности трафик $L2 \leftrightarrow L3$ формируется чтением матриц для этапа 1 ($\approx 60n/w$ байт при последовательном доступе), заполнением и вытеснением приватного буфера ($\approx 120n$ байт), чтением данных агентов — путей и оценок ($12Kn/w$ байт) и сливанием приватных буферов с глобальными матрицами, требующим чтения и записи около $180n$ байт. Суммарный объем трафика $L2 \leftrightarrow L3$ на ядро за итерацию составляет $V_{L23} \approx 300n + 6144n/w$ байт при $K = 512$. В режиме вытеснения из L3 приватные буферы находятся в DRAM. При нерегулярном доступе к ним каждая кэш-линия читается из DRAM однократно при первом промахе, а затем, будучи модифицированной, вытесняется обратно при завершении обработки блока. С учетом измеренной через аппаратные счетчики

Таблица 11. Отношение ускорения алгоритма с локальным накоплением к ускорению алгоритма без блокировок на Intel i5-12450H

Table 11. Ratio of speedup of the algorithm with local accumulation to the speedup of the algorithm without locks on Intel i5-12450H

n	w											
	1	2	3	4	5	6	7	8	9	10	11	12
42	0.96	0.72	0.63	0.82	0.70	0.68	0.71	0.69	0.72	0.71	0.72	0.69
672	0.96	0.72	0.73	0.76	0.77	0.71	0.74	0.76	0.75	0.73	0.75	0.77
10752	0.97	0.90	0.90	0.89	0.89	0.68	0.68	0.69	0.70	0.71	0.77	0.80
21504	0.96	0.98	0.92	0.93	0.96	0.90	0.86	0.85	0.86	0.88	0.90	0.92
43008	0.93	1.21	1.16	1.01	1.08	1.02	1.01	1.02	1.03	1.08	1.16	1.31
86016	0.92	1.02	1.09	1.17	1.23	1.28	1.37	1.46	1.55	1.65	1.75	1.82

доли dirty-строк $\beta \approx 0.4$ эффективный коэффициент трафика для приватного буфера составляет 1.4, что дает $84n$ байт. Чтение данных агентов добавляет $12Kn/w$ байт, а слияние с глобальными матрицами требует чтения приватного буфера из DRAM ($60n$ байт) при том, что глобальные матрицы суммарным объемом $60n$ байт остаются в L3. Итоговый трафик DRAM на ядро за итерацию: $V_{\text{DRAM}} \approx 144n + 6144n/w$ байт. За 500 итераций суммарный объем данных, проходящих через критический интерфейс, составляет $V_{\text{total}} = 500 \cdot w \cdot (V_{\text{L23}} + V_{\text{DRAM}})$.

Помимо пропускной способности критическим фактором в режиме вытеснения из L3 является латентность доступа к DRAM. Доступ к приватным буферам на этапе 3 осуществляется по индексу, определяемому псевдослучайным выбором агента, что делает паттерн доступа принципиально нерегулярным и не поддающимся аппаратной предвыборке. Каждый промах мимо L3 приводит к полной задержке обращения к DRAM. Число таких промахов на ядро за итерацию составляет $N_{\text{miss}} = 2Kn/w = 1024\ n/w$. Благодаря параллелизму уровня памяти (memory-level parallelism, MLP), измеренному через счетчик `lld_pend_miss.pending_cycles` и составляющему около 10 для микроархитектур Intel Skylake/Coffee Lake и около 8 для AMD Zen 4, эффективное число сериализованных промахов сокращается до $N_{\text{serial}} = N_{\text{miss}}/\text{MLP}$. При типичной латентности DRAM $L_{\text{DRAM}} = 80$ нс для потребительских платформ и $L_{\text{DRAM}} = 98$ нс для двухпроцессорной конфигурации Xeon 6130 (с учетом 30% удаленных NUMA-доступов) время ожидания данных за 500 итераций оценивается как $T_{\text{latency}} = 500 \cdot N_{\text{serial}} \cdot L_{\text{DRAM}}$. Результаты анализа для всех протестированных процессоров при характерных размерностях $n = 10752$ (режим L3-резидентности) и $n = 43008$ (режим вытеснения из L3) представлены в табл. 12.

В режиме L3-резидентности ($n = 10752$) эффективность использования интерфейса $L2 \leftrightarrow L3$ варьируется от 11.8% у двухпроцессорного Xeon 6130 до 27.2% у Ryzen 9 7900. Наиболее высокие показатели демонстрируют процессоры с большим объемом L3-кэша и оптимизированной кольцевой шиной: Intel Core i5-10400F достигает 24.6% благодаря 12 Мбайт L3 и эффективной архитектуре Intel Comet Lake, а AMD Ryzen 9 7900 с 64 Мбайт L3 и двумя кристаллами CCD показывает наилучший результат 27.2%. Наименьшая эффективность у Intel Xeon 6130 (11.8%) обусловлена относительно низкой тактовой частотой кольцевой шины (2.1 ГГц) и накладными расходами на когерентность кэша между двумя сокетами. Доля времени, потерянного на ожидание данных из-за промахов мимо L2, не превышает 10% для всех процессоров, причем у AMD Zen 4 этот показатель снижается до значений 2...3% благодаря увеличенному объему L2-кэша (1 Мбайт на ядро против 256 Кбайт для Intel Coffee Lake). В режиме вытеснения из L3 ($n = 43008$) эффективность использования DRAM падает до 1.85% для Intel Core i5-8300H, 5.08% для Intel Core i5-9400F, 10.6% для Intel Core i5-10400F и 14.0% для AMD Ryzen 9 7900. Доля времени ожидания данных из DRAM при этом возрастает до 54.4% для Intel Core i5-8300H, 38.7% для Intel Core i5-9400F, 29.2% для Intel Core i5-10400F и 21.7% для AMD Ryzen 9 7900. Хорошо видна прямая корреляция между объемом L3-кэша и устойчивостью к деградации: каждый дополнительный мегабайт L3 позволяет удерживать большую долю приватных буферов в кэше, снижая интенсивность обращений к DRAM. Процессор Intel Core i5-8300H с L3-кэшем объемом 8 Мбайт страдает наиболее сильно так как более половины времени выполнения уходит на ожидание данных, что объясняет катастрофическое падение ускорения до 1.70 на 4 потоках. Важно подчеркнуть, что канал DRAM не насыщен ни на одном из протестированных процессоров. Его реальная пропускная способность в 5–20 раз ниже пиковой, измеренной бенчмарком

Таблица 12. Эффективность использования подсистемы памяти для различных процессоров
 Table 12. Memory subsystem utilization efficiency for different processors

Процессор CPU model	Режим 1 ($n = 10752$) Mode 1 ($n = 10752$)			Режим 3 ($n = 43008$) Mode 3 ($n = 43008$)		
	w	$\eta_{\text{L23}}, \%$	$T_{\text{latency}}, \%$	w	$\eta_{\text{DRAM}}, \%$	$T_{\text{latency}}, \%$
Intel Core i5-8300H (8 Мбайт L3)	8	15.3	≈ 10	4	1.85	54.4
Intel Core i5-9400F (9 Мбайт L3)	6	16.4	≈ 8	6	5.08	38.7
Intel Core i5-10400F (12 Мбайт L3)	12	24.6	≈ 5	12	10.6	29.2
Intel Core i5-12450H (12 Мбайт L3)	8	15.3	≈ 9	8	6.30	35.1
AMD Ryzen 5 7500F (32 Мбайт L3)	6	17.8	≈ 3	6	8.39	28.4
AMD Ryzen 9 7900 (64 Мбайт L3)	12	27.2	≈ 2	12	14.0	21.7
Intel Xeon 6130 2S (44 Мбайт L3)	24	11.8	≈ 6	16	8.88	33.8



STREAM. Это однозначно указывает на то, что узким местом является не пропускная способность как таковая, а латентность нерегулярного доступа к данным, вытесненным в DRAM. Наблюдаемое снижение масштабируемости при больших размерностях параметрического графа не является дефектом предложенной реализации, а фундаментально обусловлено иерархией памяти современных процессоров. Любая программная реализация с аналогичным паттерном нерегулярного доступа к данным столкнется с тем же ограничением. Эффективный способ преодолеть ограничение — изменить сам алгоритм так, чтобы восстановить локальность данных и вернуть критические структуры в более быстрые уровни кэш-памяти.

5.8. Блочная модификация алгоритма с локальным накоплением. Ключевой проблемой масштабируемости матричного АСО, выявленной в разделах 5.2 и 5.6, является вытеснение частных буферов потоков из L3-кэша при больших размерностях параметрического графа ($n > 20000$). Для решения этой проблемы предложена блочная модификация алгоритма с локальным накоплением, в которой потоки работают не со всей матрицей целиком, а с ее последовательными блоками (chunks). Алгоритм работы этапа 3:

1. Матрицы $T(t)$ и $\Theta(t)$ (размерности $n \times m$) разбиваются по первому измерению на B непересекающихся блоков по $n_B = \lceil n/B \rceil$ слоев в каждом.
2. Каждый поток выделяет частные буферы не на всю матрицу ($60n$ байт), а только на один блок ($60n_B$ байт).
3. Выполняется последовательный цикл по блокам $b = 1, \dots, B$:
 - Потоки независимо обрабатывают свои назначенные слои внутри текущего блока (выполняют этап 3: добавление феромона и учет посещений от всех агентов), накапливая изменения в своих локальных буферах размера $60n_B$.
 - По завершении обработки блока производится синхронизация и слияние локальных буферов потоков с глобальными матрицами для данного блока.
 - Потоки переходят к следующему блоку.
4. После обработки всех блоков выполняется испарение феромона (поэлементное умножение на ρ), которое может быть выполнено как отдельный параллельный цикл.

Ключевое преимущество данного подхода заключается в том, что объем частных буферов на один поток сокращается с $60n$ до $60n/B$ байт. Выбирая B достаточным, чтобы $60n/B$ не превышало эффективный объем L2-кэша ядра (или, по крайней мере, чтобы $60nw/B$ не превышало объем L3-кэша), можно полностью устранить деградацию производительности, связанную с вытеснением данных в DRAM.

В табл. 13 приведены результаты экспериментального исследования блочной модификации на процессорах Intel Core i5-9400F (6 ядер, 9 Мбайт L3) и AMD Ryzen 5 7500F (6 ядер, 32 Мбайт L3). Размерность параметрического графа $n = 86016$, размер популяции $K = 512$, число блоков $B = 21$ ($n_B = 4096$, объем частного буфера на поток $60 \times 4096 \approx 240$ Кбайт, что полностью помещается в L2-кэш обоих процессоров).

Для Intel Core i5-9400F, который в стандартной реализации при $n = 86016$ показывал ускорение лишь $S_6 = 1.66$ раз (см. раздел 5.3), блочная модификация позволяет достичь $S_6 = 5.09$ раз — практически линейное ускорение на 6 физических ядрах. Для AMD Ryzen 9 7900 ускорение составляет $S_6 = 5.59$ раз (при эффективности $E_6 = 0.93$), что также значительно превосходит показатели стандартной реализации.

Таблица 13. Время выполнения и ускорение для блочной модификации алгоритма ($n = 86016$, $K = 512$, $B = 21$)
 Table 13. Time of execution and speedup of the chunked algorithm modification ($n = 86016$, $K = 512$, $B = 21$)

Процессор CPU model	Время выполнения, мс/слой Time of execution, ms/layer				Ускорение Speedup			
	Число потоков w Number of threads w				Число потоков w Number of threads w			
	1	2	4	6	1	2	4	6
Intel Core i5-9400F	4.78	2.49	1.68	0.94	1.00	1.92	2.84	5.09
Intel Core i5-12450H	4.11	2.15	1.17	1.04	1.00	1.91	3.51	3.96
AMD Ryzen 9 7900	3.207	1.599	0.816	0.570	1.00	2.00	3.92	5.63

Таблица 14. Время выполнения блочной модификации алгоритма для процессора AMD Ryzen 9 7900 при различных размерностях n ($B = 21$), мс/слойTable 14. Time of execution of the chunked algorithm modification for AMD Ryzen 9 7900 for different n ($B = 21$), ms/layers

n	w												
	1	2	4	6	8	10	12	14	16	18	20	22	24
42	4.423	2.333	1.379	1.160	1.699	1.885	2.077	1.905	2.018	2.233	2.391	2.581	2.742
672	3.247	1.633	0.839	0.592	0.527	0.475	0.447	0.451	0.449	0.437	0.446	0.471	0.477
2688	3.227	1.619	0.832	0.577	0.481	0.416	0.377	0.343	0.319	0.295	0.309	0.318	0.329
10752	3.200	1.609	0.820	0.569	0.465	0.399	0.371	0.355	0.342	0.332	0.338	0.346	0.361
43008	3.201	1.602	0.818	0.570	0.469	0.407	0.374	0.355	0.341	0.331	0.342	0.350	0.365
86016	3.207	1.600	0.816	0.574	0.466	0.407	0.370	0.359	0.356	0.325	0.339	0.357	0.366
172032	3.197	1.596	0.818	0.578	0.464	0.398	0.364	0.356	0.348	0.355	0.366	0.372	0.393

При использовании всех 12 физических ядер (табл. 14) для размерности $n = 86016$ ускорение достигает $S_{12} = 8.67$, а при $n = 172032$ — $S_{12} = 8.78$ раз.

5.9. Выбор оптимального числа блоков. Ключевым параметром блочной модификации является число блоков B (или, эквивалентно, размер блока $n_B = \lceil n/B \rceil$). Оптимальное число блоков B_{opt} должно выбираться с учетом следующих двух условий.

1. Условие L2-резидентности приватного буфера (для максимальной производительности):

$$n_B \leq \frac{V_{L2}^{\text{core}}}{60} \Rightarrow B \geq \frac{60n}{V_{L2}^{\text{core}}}, \quad (2)$$

где V_{L2}^{core} — объем L2-кэша одного ядра в байтах.

2. Условие L3-резидентности суммарных приватных буферов (для предотвращения вытеснения в DRAM):

$$n_B \leq \frac{V_{L3}}{60w} \Rightarrow B \geq \frac{60nw}{V_{L3}}. \quad (3)$$

На практике решающим является первое условие, так как L2-кэш значительно меньше L3, а нерегулярный доступ к приватным буферам на этапе 3 делает их L2-резидентность критической для производительности. Для процессоров с объемом L2-кэша 256 КБ (Intel Core i5-8300H, Intel Core i5-9400F, Intel Core i5-10400F) условие (2) дает:

$$n_B \leq \frac{256 \times 1024}{60} \approx 4369 \Rightarrow B \geq \frac{n}{4369}.$$

Для $n = 86016$ это дает $B \geq 19.7$, т.е. $B = 20$ или 21, что полностью согласуется с экспериментальными данными.

Для процессоров с L2-кэшем 1 Мбайт (Intel Core i5-12450H P-core, AMD Ryzen 5 7500F, AMD Ryzen 9 7900) условие (2) дает:

$$n_B \leq \frac{1024 \times 1024}{60} \approx 17476 \Rightarrow B \geq \frac{n}{17476}.$$

Это означает, что для $n = 86016$ достаточно $B \geq 5$, а для $n = 43008$ — $B \geq 3$. Однако на практике следует выбирать B несколько больше расчетного минимума, чтобы учесть конкуренцию за L2-кэш со стороны других данных (матрицы вероятностей $F(t)$, кода самого алгоритма) и, для гибридных архитектур, меньший объем L2-кэша Е-ядер.

Рекомендуемая стратегия выбора B_{opt} :

1. Рассчитать минимальное B_{min} по формуле (2) с запасом 20...30%: $B_{\text{rec}} = \lceil 1.3 \cdot 60n/V_{L2}^{\text{core}} \rceil$.
2. Увеличить B_{rec} , если количество потоков w велико, чтобы также удовлетворить условию (3).
3. Ограничить B_{rec} сверху значением $\lceil n/32 \rceil$, чтобы избежать чрезмерных накладных расходов на синхронизацию (при $n_B < 32$ эффективность падает).

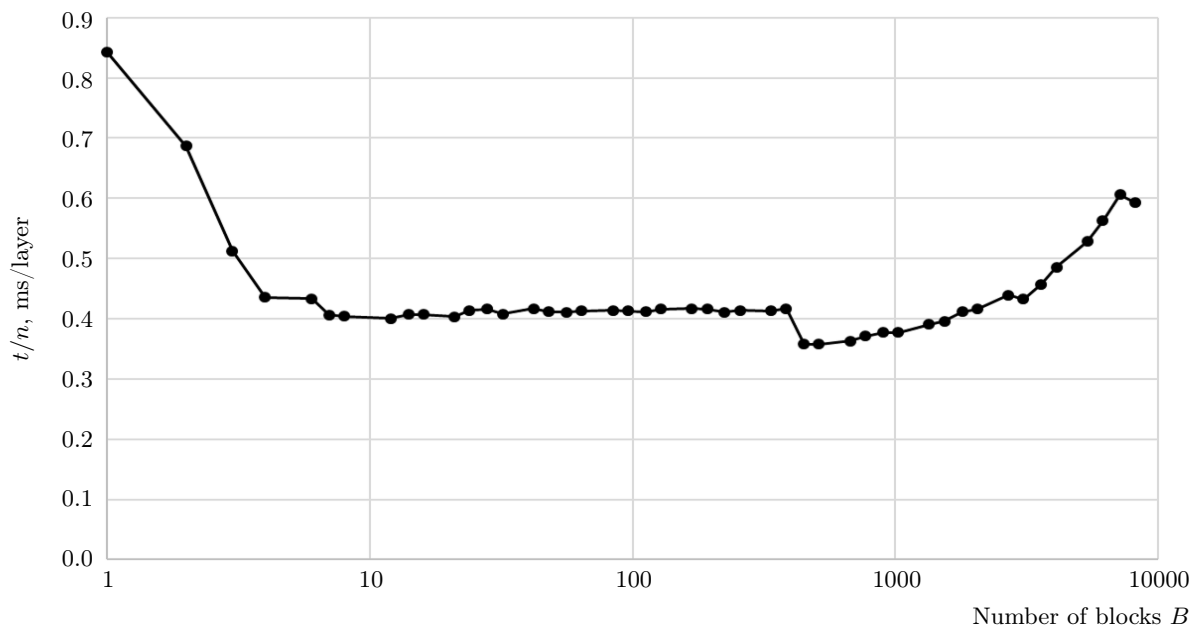


Рис. 2. Зависимость времени выполнения (мс/слой) от числа блоков B для AMD Ryzen 9 7900 ($n = 172032$, $K = 512$, $w = 24$)

Fig. 2. Dependence of time (ms/layer) on the number of blocks B for AMD Ryzen 9 7900 ($n = 172032$, $K = 512$, $w = 24$)

Проведено исследование зависимости времени выполнения от B на процессоре AMD Ryzen 9 7900 при $n = 172032$, $K = 512$, $w = 24$ (рис. 2).

При $B < 21$ время выполнения превышает 0.40 мс/слой из-за неэффективного использования кэш-линий при большом размере блока. В диапазоне $21 \leq B \leq 2048$ время стабилизируется на минимальном уровне 0.40...0.42 мс/слой, что соответствует L2-резидентности приватных буферов ($n_B \leq 8192$, объем буфера ≤ 491 Кбайт < 1 Мбайт L2-кэша, имеющегося в микроархитектуре AMD Zen 4). При $B > 2048$ время начинает расти, достигая 0.61 мс/слой при $B = 7168$, из-за накладных расходов на синхронизацию между блоками. Предложенная блочная модификация позволяет эффективно масштабировать матричный АСО на задачи практически любой размерности, ограниченной лишь доступным объемом оперативной памяти, а не размером кэша.

Доля полезной работы (вычисленная по показателю Retiring vtune) для процессора Intel Core i5-12450H в стандартной реализации при $n = 10752$, $w = 12$ составляет 16.4%, но при переходе к $n = 86016$ этот показатель падает до 7.6% (снижение в 2.16 раза) (табл. 15). Данный показатель означает, что более 92% времени процессор простаивает в ожидании данных из DRAM. Для блочной модификации ($B = 21$) при $n = 86016$, $w = 12$ доля полезной работы восстанавливается до 16.6%, что соответствует уровню кэш-резидентного режима, при этом данный показатель практически не зависит от числа потоков. Блочная модификация не только снижает DRAM Bound и CPI, но и восстанавливает долю полезной работы процессора до уровня, характерного для кэш-резидентного режима.

Таблица 15. Сравнение доли полезной работы для стандартной и блочной реализаций при $w = 12$

Table 15. Comparison of useful work fraction for standard and chunked implementations at $w = 12$

Реализация Implementation	$n = 10752$	$n = 21504$	$n = 43008$	$n = 86016$
Стандартная Conventional	16.4%	15.0%	11.0%	7.6%
Блочная ($B = 21$) Block-based ($B = 21$)	16.8%	16.8%	16.8%	16.6%

6. Заключение. В работе представлен полный цикл исследования матричной модификации метода муравьиных колоний для параметрической оптимизации.

Основные результаты:

- Выполнен переход от графовой модели АСО к матричному формализму, доказано снижение асимптотической сложности с $O(K \cdot n^2)$ до $O(K \cdot n)$.
- На основе анализа арифметической интенсивности ($\nu_{\max} \approx 0.44$ оп/байт) алгоритм строго отнесен к классу memory-bound.
- Разработана иерархическая модель кэш-резидентности, предсказывающая четыре режима масштабируемости, подтвержденные экспериментально на 7 процессорах различных архитектур.
- Выявлены паттерны масштабирования на гибридной архитектуре Intel Alder Lake (P-core, E-core, Hyper-Threading) и даны рекомендации по применению привязки потоков к ядрам.
- Обнаружен эффект инверсии эффективности при отказе от локальных буферов (до 1.82 раз при $n = 86016$).
- Предложена блочная модификация алгоритма, обеспечивающая практически линейное ускорение ($S_6 = 5.09$ на Intel Core i5-9400F) за счет управления L2-резидентностью частных буферов. Разработаны аналитические соотношения для выбора оптимального числа блоков B_{opt} .

Полученные результаты формируют теоретическую и практическую базу для разработки высоко-масштабируемых метаэвристических методов оптимизации на платформах с общей памятью и открывают перспективы для решения параметрических задач, основанных на графах больших размерностей [24].

Список литературы

1. Colormi A., Dorigo M., Vittorio M. Distributed Optimization by Ant Colonies // Proceedings of the First European Conference on Artificial Life. Elsevier Publishing. 1991. 134–142.
2. Dorigo M., Gambardella L.M. Ant Colony System: a Cooperative Learning Approach to the Traveling Salesman Problem // Evolutionary Computation, IEEE Trans. 1997. 1.1. 53–66. doi 10.1109/4235.585892.
3. Dorigo M., Maniezzo V., Colormi A. Positive Feedback as a Search Strategy // Technical Report 91-016, Politecnico di Milano, Italy, 1991.
4. Gambardella L.M., Dorigo M. Ant-Q: A Reinforcement Learning Approach to the Traveling Salesman Problem // Machine Learning Proceedings 1995. 252–260. doi 10.1016/B978-1-55860-377-6.50039-6.
5. Bullnheimer B., Hartl R.F., Strauss C. Applying the Ant System to the Vehicle Routing Problems // Annals of Operations Research. 1998. 79. 109–123.
6. Stützle T., Hoos H.H. MAX-MIN Ant System // Future Generation Computer Systems. 2000. 16, N 8. 889–914. doi 10.1016/S0167-739X(00)00043-1.
7. Cordon O.G., de Viana I.F., Herrera F. Analysis of the Best-Worst Ant System and Its Variants on the QAP // Third International Workshop, ANTS 2002, Brussels, Belgium, September 12–14, 2002. Springer Berlin Heidelberg, 2002, pp. 228–234. doi 10.1007/3-540-45724-0_20.
8. Randall M., Lewis A.A. A Parallel Implementation of Ant Colony Optimization // Journal of Parallel and Distributed Computing. 2002. 62, N 9. 1421–1432. doi 10.1006/jpdc.2002.1854.
9. Dorigo M., Stützle T. Ant Colony Optimization // Cambridge, Massachusetts: MIT Press. 2004, 321
10. Bai H., OuYang D., Li X., et al. MAXMIN Ant System on GPU with CUDA // 2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC). IEEE Computer Society. 2009. 801–804. doi 10.1109/ICICIC.2009.255.
11. Chitty D.M. Applying ACO to Large Scale TSP Instances // F. Chao et al. (Eds.) Advances in Computational Intelligence Systems. UKCI. Intelligent Systems and Computing, Springer, Cham. 2018. 650. doi 10.1007/978-3-319-66939-7_9.
12. Skinderowicz R. The GPU-based parallel Ant Colony System // Journal of Parallel and Distributed Computing. 2016. 98. 48–60. doi 10.1016/j.jpdc.2016.04.014.
13. Zhang D., You X., Liu S., Yang K. Multi-Colony Ant Colony Optimization Based on Generalized Jaccard Similarity Recommendation Strategy // IEEE Access. 2019. 7. 157303–157317. doi 10.1109/ACCESS.2019.2949860.
14. Skinderowicz R. Implementing a GPU-based parallel MAX-MIN Ant System // Future Generation Computer Systems. 2020. 106. 277–295. doi 10.1016/j.future.2020.01.011.



15. Huan Z.B., Fu G.T., Fa T.H., et al. High Performance Ant Colony System Based on GPU Warp Specialization with a Static-Dynamic Balanced Candidate Set Strategy // *Future Generation Computer Systems*. 2021. **125**. 136–150. doi 10.1016/j.future.2021.06.041.
16. Sinitsyn I.N., Titov Y.P. Control of Set of System Parameter Values by the Ant Colony Method // *Autom. Remote Control*. 2023. **84**. 893–903. doi 10.1134/S0005117923080106.
17. Sudakov V.A., Titov Y.P. Matrix-Based ACO for Solving Parametric Problems Using Heterogeneous Reconfigurable Computers and SIMD Accelerators // *Mathematics*. 2025. **13** № 1284. doi 10.3390/math13081284.
18. Судakov В. А., Титов Ю.П. Исследование модели параметрического графа в методе муравьиных колоний // *Математическое моделирование*. 2024. **36**. №6. 21–37. doi 10.20948/mm-2024-06-02.
19. Федотов И.Е. Модели параллельного программирования. М.: СОЛОН-ПРЕСС, 2012.
20. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002.
21. Синицын И.Н., Титов Ю.П. Исследование алгоритмов циклического поиска дополнительных решений при оптимизации порядка следования гиперпараметров методом муравьиных колоний // *Системы высокой доступности*. 2023. **19**, №1. 59–73. doi 10.18127/j20729472-202301-05.
22. ACO_SIMD/OMP C++ Optimal at main · kalengul/ACO_SIMD · GitHub, https://github.com/kalengul/ACO_SIMD/tree/main/OMP%20%D0%A1%2B%2B%20%20Optimal. Cited September 21, 2026.
23. Mishra S.K. Some New Test Functions for Global Optimization and Performance of Repulsive Particle Swarm Method // *University Library of Munich, Germany, MPRA Paper*. 2006. <https://mpra.ub.uni-muenchen.de/2718/>. Cited February 25, 2026.
24. Chetverushkin B.N., Sudakov V.A., Titov Y.P. Graph Condensation for Large Factor Models // *Dokl. Math*. 2024. **109**. 246–251. doi 10.1134/S1064562424702090.

Получена
4 июня 2026 г.

Принята
10 августа 2026 г.

Опубликована
1 октября 2026 г.

Информация об авторах

Юрий Павлович Титов — к.т.н., доцент; Московский авиационный институт (национальный исследовательский университет), Волоколамское шоссе, д. 4, 125080, Москва, Российская Федерация.

Константин Александрович Кордовер — Московский авиационный институт (национальный исследовательский университет), Волоколамское шоссе, д. 4, 125080, Москва, Российская Федерация.

Павел Александрович Жданов — Московский авиационный институт (национальный исследовательский университет), Волоколамское шоссе, д. 4, 125080, Москва, Российская Федерация.

Александр Александрович Жданов — Московский авиационный институт (национальный исследовательский университет), Волоколамское шоссе, д. 4, 125080, Москва, Российская Федерация.

References

1. A. Coloni, M. Dorigo and M. Vittorio, “Distributed Optimization by Ant Colonies,” *Proceedings of the First European Conference on Artificial Life. Paris, France, January 1991* (Elsevier Publishing, 1991), pp. 134–142.
2. M. Dorigo and L. M. Gambardella, “Ant Colony System: a Cooperative Learning Approach to the Traveling Salesman Problem,” *Evolutionary Computation, IEEE Transactions*. **1.1**, 53–66 (1997). doi 10.1109/4235.585892.
3. M. Dorigo, V. Maniezzo and A. Coloni, “Positive Feedback as a Search Strategy,” In *Technical Report 91-016, Politecnico di Milano, Italy, 1991*.
4. L. M. Gambardella and M. Dorigo, “Ant-Q: A Reinforcement Learning Approach to the Traveling Salesman Problem,” In *Machine Learning Proceedings 1995, Morgan Kaufmann*, pp. 252–260. doi 10.1016/B978-1-55860-377-6.50039-6.
5. B. Bullnheimer, R. F. Hartl and C. Strauss, “Applying the Ant System to the Vehicle Routing Problems,” *Annals of Operations Research*. **79**, 109–123 (1998). doi 10.1007/978-1-4615-5775-3_20.
6. T. Stützle and H. H. Hoos, “MAX-MIN Ant System,” *Future Generation Computer Systems*. **16** (8), 889–914 (2000). doi 10.1016/S0167-739X(00)00043-1.

7. O. G. Cordon, I. F. de Viana, and F. Herrera, “Analysis of the Best-Worst Ant System and Its Variants on the QAP,” Proc. in *Third International Workshop, ANTS 2002*, M. Dorigo, G. Di Caro, M. Sampels (eds), Brussels, Belgium, September 12–14, 2002 (Springer Berlin Heidelberg, 2002), pp. 228–234. doi [10.1007/3-540-45724-0_20](https://doi.org/10.1007/3-540-45724-0_20).
8. M. Randall, A. A. Lewis, “A Parallel Implementation of Ant Colony Optimization,” *Journal of Parallel and Distributed Computing*. **62** (9), 1421–1432 (2002). doi [10.1006/jpdc.2002.1854](https://doi.org/10.1006/jpdc.2002.1854).
9. M. Dorigo, T. Stützle, *Ant Colony Optimization* (MIT Press, Cambridge, Massachusetts, 2004).
10. H. Bai, D. OuYang, X. Li, et al., “MAXMIN Ant System on GPU with CUDA,” In *2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC)*, Kaohsiung, Taiwan, December 7–9, 2009 (IEEE Computer Society), pp. 801–804. doi [10.1109/ICICIC.2009.255](https://doi.org/10.1109/ICICIC.2009.255).
11. D. M. Chitty, “Applying ACO to Large Scale TSP Instances,” In: *F. Chao et al. (Eds.) Advances in Computational Intelligence Systems. UKCI 2017* (Intelligent Systems and Computing, Springer, Cham), **650** (2018). doi [10.1007/978-3-319-66939-7_9](https://doi.org/10.1007/978-3-319-66939-7_9).
12. R. Skinderowicz, “The GPU-based parallel Ant Colony System,” *Journal of Parallel and Distributed Computing*. **98**, 48–60 (2016) doi [10.1016/j.jpdc.2016.04.014](https://doi.org/10.1016/j.jpdc.2016.04.014).
13. D. Zhang, X. You, S. Liu, K. Yang, “Multi-Colony Ant Colony Optimization Based on Generalized Jaccard Similarity Recommendation Strategy,” *IEEE Access*. **7**, 157303–157317 (2019) doi [10.1109/ACCESS.2019.2949860](https://doi.org/10.1109/ACCESS.2019.2949860).
14. R. Skinderowicz, “Implementing a GPU-based parallel MAX–MIN Ant System,” *Future Generation Computer Systems*. **106**, 277–295 (2020) doi [10.1016/j.future.2020.01.011](https://doi.org/10.1016/j.future.2020.01.011).
15. Z. B. Huan, G. T. Fu, T. H. Fa, et al., “High Performance Ant Colony System Based on GPU Warp Specialization with a Static-Dynamic Balanced Candidate Set Strategy,” *Future Generation Computer Systems*. **125**, 136–150 (2021). doi [10.1016/j.future.2021.06.041](https://doi.org/10.1016/j.future.2021.06.041).
16. I. N. Sinitsyn, Y. P. Titov, “Control of Set of System Parameter Values by the Ant Colony Method,” *Autom. Remote Control*, **84**, 893–903 (2023). doi [10.1134/S0005117923080106](https://doi.org/10.1134/S0005117923080106).
17. V. Sudakov, Y. Titov, “Matrix-Based ACO for Solving Parametric Problems Using Heterogeneous Reconfigurable Computers and SIMD Accelerators,” *Mathematics*, **13** (1284), (2025). doi [10.3390/math13081284](https://doi.org/10.3390/math13081284).
18. V. A. Sudakov, Y. P. Titov, “Investigation of the Parametric Graph Model in the Ant Colony Method,” *Math. Models Comput. Simul.* **17**, 126–136 (2025). doi [10.1134/S2070048224700996](https://doi.org/10.1134/S2070048224700996).
19. I. E. Fedotov, *Models of Parallel Programming* (SOLON-Press, Moscow, 2012) [in Russian].
20. V. V. Voevodin and V. V. Voevodin, *The Parallel Computing* (BHV-Petersburg, St. Petersburg, 2002) [in Russian].
21. I. N. Sinitsyn, Y. P. Titov, “Investigation of algorithms for cyclic search for additional solutions when optimizing the order of hyperparameters by the ant colony method,” *High Availability Systems*. **19** (1), 59–73 (2023). http://raidiotec.ru/en/journal/Highly_available_systems/number/2023-1/article/23354. Cited September 21, 2026.
22. ACO_SIMD/OMP C++ Optimal at main · kalengul/ACO_SIMD · GitHub, https://github.com/kalengul/ACO_SIMD/tree/main/OMP%20D0%A1%2B%2B%20%20Optimal. Cited September 21, 2026.
23. S. K. Mishra, “Some New Test Functions for Global Optimization and Performance of Repulsive Particle Swarm Method,” *University Library of Munich, Germany, MPRA Paper*. **2718** (2006). <https://mpra.ub.uni-muenchen.de/2718/>. Cited September 21, 2026.
24. B. N. Chetverushkin, V. A. Sudakov, Y. P. Titov, “Graph Condensation for Large Factor Models,” *Dokl. Math.* **109**, 246–251 (2024). doi [10.1134/S1064562424702090](https://doi.org/10.1134/S1064562424702090).

Received
June 4, 2026

Accepted
August 10, 2026

Published
October 1, 2026

Information about the authors

Yurii P. Titov — Ph. D., Associate Professor; Moscow Aviation Institute (National Research University), Volokolamskoe shosse, 4, 125080, Moscow, Russia.

Konstantin A. Kordover — Moscow Aviation Institute (National Research University), Volokolamskoe shosse, 4, 125080, Moscow, Russia.

Pavel A. Zhdanov — Moscow Aviation Institute (National Research University), Volokolamskoe shosse, 4, 125080, Moscow, Russia.

Alexander A. Zhdanov — Moscow Aviation Institute (National Research University), Volokolamskoe shosse, 4, 125080, Moscow, Russia.