CODE OPTIMISATION ON THE EXAMPLE OF AN ALGORITHM FOR SOLVING THE TRAVELING SALESMAN PROBLEM

Yu. F. Leonova

National Research South Ural State University,
454080, Chelyabinsk, Russia

DOI: 10.24412/2073-0667-2025-2-48-64
EDN: KQAAFR

This paper presents a comprehensive approach to optimizing the cycle merging algorithm applied to the Traveling Salesman Problem (TSP), a classic NP-hard problem that has challenged researchers and practitioners alike in logistics, manufacturing, and data-intensive applications. The TSP requires finding the shortest possible route that visits a list of cities and returns to the starting point. As the number of cities grows, finding an exact solution becomes computationally prohibitive, making approximation techniques both necessary and valuable in practical applications.

The cycle merging algorithm is a well-established heuristic approach to solving TSP. It constructs an initial 2-factor solution that includes a set of cycles covering all vertices, and iteratively merges the cycles based on optimal edge replacement until only one cycle remains. Along with the choice of the solution algorithm, the quality of the application code plays an important role.

In the process of work, a number of measures aimed at optimising the program code implementing the cycle merging algorithm have been performed. The approach includes optimising the algorithm, optimising the data storage structure and using parallel programming techniques.

Experimental results show that the optimised algorithm significantly out-performs the baseline implementation, achieving a speedup factor proportional to the number of computational cores and nodes. Tests conducted on instances with up to 1000 nodes showed that our approach makes it possible to solve larger problems without a commensurate increase in computational resources. The study also observed a consistent performance gain in cache utilisation and a reduction in latency at key stages of the algorithm, which confirms the effectiveness of the chosen optimisations.

This work provides a sound basis for solving large TSP instances by combining heuristic methods with advanced computational optimisations. The results highlight the importance of both algorithm efficiency and imple-mentation techniques when solving computationally intensive problems. The approach and results presented here are not only applicable to TSP, but also to a broader class of combinatorial optimisation problems where parallelism and memory efficiency are important. Future work may investigate additional optimisations through GPU acceleration or hybrid parallelism techniques, potentially providing even better performance.

Key words: traveling salesman problem, combinatorial optimisation, software code optimisation, performance optimisation, parallel computing, instrumentation and profiling.

References

1. Jarrah A., Bataineh A. S. A., Almomany A. The optimisation of travelling salesman problem based on parallel ant colony algorithm // International Journal of Computer Applications in Technology. 2022. V. 69. N 4. P. 309–321.

2. Rhee Y. Gpu-based parallel ant colony system for traveling salesman problem // Journal of The Korea Society of Computer and Information. 2022. V. 27. N 2. P. 1–8.

3. Wang Z. et al. A fine-grained fast parallel genetic algorithm based on a ternary optical computer for solving traveling salesman problem // The Journal of Supercomputing. 2023. V. 79. N 5. P. 4760–4790.

4. Peng C. Parallel genetic algorithm for travelling salesman problem // In: International conference on automation control, algorithm, and intelligent bionics (ACAIB 2022). 2022. V. 12253, P. 259–267.

5. Alhenawi E. et al. Solving Traveling Salesman Problem Using Parallel River Formation Dynamics Optimization Algorithm on Multi-core Architecture Using Apache Spark // International Journal of Computational Intelligence Systems. 2024. V. 17. N 1. P. 4.

6. Qiao Y. et al. A hybridized parallel bats algorithm for combinatorial problem of traveling salesman // Journal of Intelligent & Fuzzy Systems. 2020. T. 38. N 5. P. 5811–5820.

7. Korol Z. A., Ankudinov K. A., Korolkova L. N. Research of the methods of software code optimisation to improve performance // Innovative directions of development in education, economics, engineering and technology. 2023. P. 257–260. (in Russian).

8. Taik A. M., Lupin S. A. A., Fedyashin D. A. MPI library usage for parallel realisation of the algorithm of the complete variant search // Software Products and Systems. 2023. V 36. N 4. P. 607–614. (in Russian).

9. Panyukov A. V., Leonova Y. F. Algorithm for approximate solution of the travelling salesman problem // Optimization Problems and Their Applications (OPTA-2018). 2018. P. 31. (in Russian).

10. Leonova Yu. F. Cycles merging algorithm for an approximate solution of the traveling salesman problem // Abstracts of the XIX All-Russian Conference of Young Scientists on Mathematical Modeling and Information Technologies, Novosibirsk: ICT SB RAS. 2018. P. 28.

11. Panyukov A. V., Leonova Yu. F. Cycle Merging Algorithm for MAX TSP Problems // XVIII International Conference "Mathematical Optimization Theory and Operations Research" (MOTOR 2019), Ekaterinburg, Russia: Publisher "UMC UrFU". 2019. P. 57.

12. Panyukov A. V., Leonova Yu. F. Cycle merging algorithm for the maximal metric traveling salesman problem // Bulletin of the South Ural State University, V. 10, N 4: a series of Computational Mathematics and Informatics. Chelyabinsk: Publishing House of SUSU. 2021. P. 26–36. (in Russian) DOI: 10.14529/cmse210402.

13. Certificate of state registration of computer programme N 2021669214 Russian Federation. Programme for implementing the cycle merging algorithm for solving the travelling salesman problem : № 2021668239 : applied. 16.11.2021: published 25.11.2021 / Yu. F. Leonova, A. V. Panyukov ; applicant Federal State Autonomous Educational Institution of Higher Education 'South Ural State University'. - EDN RPNSPO.

14. Guntheroth, K. Optimized C++: Proven Techniques for Heightened Performance, O'Reilly Media. 2016.

15. Panyukov A. V., Telegin V. A. Technique of software implementation of streaming algorithms // Bulletin of South Ural State University. Series: Mathematical modelling and programming. 2008. N 27 (127). P. 78–99 (in Russian).

16. 17. Leonova Yu. Parallel implementation of the cycle merging algorithm for solving the traveling salesman problem. [Electron. Res.]: `https://github.com/YuliyaLeonova/CycleMergingAlgorithm.git`, last accessed 2025/05/19.

18. Meyers, S. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. 2018. 304 p.

19. std::mersenne_twister_engine. [Electron. Res.]: https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine.html, last accessed 2024/11/02.

20. Best Practices in the Parallel Patterns Library. [Electron. Res.]: https://learn.microsoft.com/en-us/cpp/parallel/concrt/best-practices-in-the-parallel-patterns-library?view=msvc-170, last accessed 2024/10/15.

21. Intel VTune Profiler. [Electron. Res.]: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html/gs.hfjczc, last accessed 2024/11/11.

ОПТИМИЗАЦИЯ ПРОГРАММНОГО КОДА НА ПРИМЕРЕ АЛГОРИТМА ДЛЯ РЕШЕНИЯ ЗАДАЧИ КОММИВОЯЖЕРА

Ю. Ф. Леонова

Южно-Уральский государственный университет,
454080, Челябинск, Россия

УДК 004.051
DOI: 10.24412/2073-0667-2025-2-48-64
EDN: KQAAFR

В статье рассматриваются методы оптимизации программного кода, реализующего алгоритм соединения циклов для решения задачи коммивояжера. Особое внимание уделено параллелизации вычислений с использованием технологий OpenMP и MPI, а также оптимизации структуры данных и управления памятью для повышения эффективности алгоритма. Применение предложенных методов к задачам большой размерности продемонстрировало значительное сокращение времени выполнения и более эффективное использование вычислительных ресурсов. Предложенные подходы к оптимизации могут быть применены к широкому кругу задач комбинаторной оптимизации, где важны быстродействие и рациональное распределение ресурсов.

Ключевые слова: задача коммивояжера, комбинаторная оптимизация, оптимизация программного кода, оптимизация производительности, параллельные вычисления, инструментирование и профилирование.

Введение. Задача коммивояжера (ЗК) — классическая NP-трудная задача дискретной оптимизации, заключающаяся в нахождении кратчайшего замкнутого маршрута по множеству городов, при условии, что каждый город посещается ровно один раз. Задача коммивояжера находит широкое применение в различных областях: в логистике, планировании маршрутов, а также в задачах анализа данных и машинного обучения.

С увеличением числа городов задача коммивояжера становится вычислительно сложной, поскольку количество возможных решений экспоненциально возрастает с увеличением размерности задачи. Оценочная вычислительная сложность задачи коммивояжера для точных методов составляет $O(n!)$, что делает невозможным использование таких методов для задач большой размерности.

Практические задачи требуют быстрых решений, поэтому вместо точных методов часто применяются различные приближенные методы решения задачи коммивояжера, которые позволяют за приемлемое время получить результат с высокой степенью точности.

Наряду с выбором алгоритма решения важную роль играет качество программного кода приложения. Вопрос оптимизации программного кода, в том числе средствами распараллеливания, активно разрабатывается в последние годы. Наиболее распространенными параллельными версиями эвристических алгоритмов для решения задачи коммивояжера являются оптимизация муравьиной колонии [1, 2] и генетические алгоритмы [3, 4]. Также

популярным направлением исследований является параллельная реализация новых метаэвристических [5] и гибридных подходов [6]. Кроме того, имеют место обзорные статьи, посвященные методам оптимизации программного кода [7]. Популярны статьи, описывающие применение конкретной технологии оптимизации, зачастную распараллеливания, к решению комбинаторных задач [8].

В рамках данной работы рассматриваются возможности оптимизации программной реализации алгоритма соединения циклов для решения задачи коммивояжера — приближенного алгоритма с доказанными оценками вычислительной сложности и точности, хорошо показавшего себя на различных классах матрицы стоимостей задачи коммивояжера.

Статья построена следующим образом: сначала дается описание алгоритма, подлежащего оптимизации, далее рассматриваются методы оптимизации программного кода, применимые к данному алгоритму, потом дается описание вычислительного эксперимента и приводится анализ полученных результатов.

1. Алгоритм соединения циклов. В данном разделе приводится описание алгоритма соединения циклов (Cycle Merging Algorithm — CMA) [9–11] для решения задачи коммивояжера, возможности оптимизации программного кода которого будут рассмотрены ниже.

Пусть $G = (V,E)$ — полный неориентированный граф. Пусть $w: \quad E \to Z^+$ — заданная весовая функция. Пусть $w(E\prime) = \sum_{e \in E\prime} w(e)$ для любого $E\prime \subset E$. Гамильтонов цикл — это цикл, проходящий через каждую вершину графа ровно один раз. Задача коммивояжера состоит в том, чтобы найти гамильтонов цикл H G с экстремальным весом $w(H)$.

Суть алгоритма соединения циклов для решения этой задачи заключается в последовательном объединении циклов экстремальных 2-факторов графа $G = (V,E)$, где V — множество вершин, E — множество ребер.

На первом шаге алгоритма в заданном графе G находится 2-регулярный суграф экстремального веса, т. е. цикловое покрытие графа. Данную конструкцию принято называть 2-фактором экстремального веса. Цикловое покрытие экстремального веса может быть найдено за время $O(n^3)$ с помощью алгоритмов задачи о назначении (АР) для соответствующего взвешенного полного двудольного графа.

На втором шаге осуществляется проверка единственности цикла полученного решения задачи о назначении. Если 2-фактор представлен несколькими циклами, то для каждой пары циклов рассчитывается экстремальная стоимость соединения.

Затем пара циклов с экстремальной стоимостью соединения заменяется объединенным циклом. Алгоритм завершает работу, когда текущий 2-фактор содержит один цикл.

Приведем псевдокод алгоритма для наглядности.

Алгоритм 1. Алгоритм соединения циклов.
Input: полный граф $G = (V, E)$, весовая функция $W : E \to R$, тип экстремума $\text{ext} \in \{\min, \max\}$
Output: гамильтонов цикл экстремального веса
Найти 2-фактор $C = \{c_1, c_2, \ldots, c_n\}$ экстремального веса;
if $n = 1$ **then**
  return C — найденное решение;
foreach *пара циклов* $(r, t) \in C$ **do**
  Найти оптимальную замену ребер для соединения r и t с минимизацией (или максимизацией) прироста веса;
while $n > 1$ **do**
  Выбрать пару циклов (r^*, t^*), соединение которых дает экстремальный вес;
  Построить новый цикл s из вершин и ребер r^* и t^*;
  Обновить 2-фактор C: добавить s и удалить r^* и t^*;
  Уменьшить n на 1;
  foreach *цикл* $t \in C$, $t \neq s$ **do**
    Пересчитать оптимальные соединения s и t;
return C — найденное решение;

Для алгоритма соединения циклов были сформулированы и доказаны следующие теоремы.

Теорема 1. Вычислительная сложность алгоритма соединения циклов не превышает $O(|V|^3)$.

Теорема 2. Пусть W_{opt} — оптимальное значение метрической задачи коммивояжера на максимум, W_C — вес максимального 2-фактора данной задачи, W_{alg} — вес цикла, построенного алгоритмом СМА. Тогда $W_{alg}/W_{opt} \geq 5/6$. Оценка $W_{alg}/W_{opt} \geqslant W_{alg}/W_C = 5/6$ достижима.

Доказательства обеих теорем приведены в [12].

2. Оптимизация программного кода алгоритма соединения циклов. В рамках данной статьи ниже будет рассматриваться код программной реализации алгоритма соединения циклов для решения задачи коммивояжера *на минимум*. Этот код легко модифицируется для решения задачи коммивояжера на максимум путем умножения всех элементов матрицы стоимостей на -1 и выбора ребер с максимальной стоимостью соединения на четвертом шаге алгоритма.

В качестве «базовой» используется реализация алгоритма, написанная А. В. Панюковым и Ю. Ф. Леоновой, прошедшая государственную регистрацию программы для электронных вычислительных машин или базы данных [13]. Этот код был написан несколько лет назад, поэтому сейчас является устаревшим и неэффективным.

Для решения задачи коммивояжера с высокой эффективностью необходима тщательная оптимизация всех аспектов реализации алгоритма, от структуры данных до параллелизации. Рассмотрим ключевые подходы, которые позволяют улучшить производительность и масштабируемость алгоритма.

2.1. *Оптимизация алгоритма.* В первую очередь на скорость выполнения программы влияет архитектура алгоритма [14]. В данной статье рассматривается конкретный алго-

Таблица 1

Сравнение алгоритмов решения задачи о назначении

	Класс Transport		Жадный алгоритм		Венгерский алгоритм	
n	Стоимость	Время	Стоимость	Время	Стоимость	Время
100	16011	0,0517	264619	0,0012	16011	0,0322
200	15861	0,7002	507938	0,0039	15861	0,1644
300	16528	5,8183	779733	0,0142	16528	0,6800
400	16504	20,7321	1050630	0,0268	16504	1,3509
500	16743	38,4814	1288700	0,0256	16743	1,6650

ритм — алгоритм соединения циклов, поэтому общая структура остается неизменной. Тем не менее, некоторые шаги алгоритма могут решаться различными способами.

Проведенный анализ производительности показал, что большую часть времени работы алгоритма занимает решение задачи о назначении. В базовой реализации алгоритма задача о назначении решается с помощью класса Transport [15]. В табл. 1 представлены результаты сравнения качества и скорости работы данного класса с жадным и венгерским алгоритмами для решения задачи о назначении. Время в таблице представлено в секундах, n — число вершин. Очевидно, что качество и скорость нахождения гамильтонова цикла зависят от качества полученного решения задачи о назначении, поэтому, несмотря на то, что жадный алгоритм работает намного быстрее, он был отвергнут, так как его точность уступает точности двух других алгоритмов. Класс Transport затрачивает существенно больше времени при идентичной длине найденного циклового покрытия. В связи с этим для дальнейшего использования был выбран венгерский алгоритм решения задачи о назначении.

2.2. *Оптимизация структуры данных.* Одним из важных аспектов оптимизации является выбор эффективных структур данных для представления графа и матрицы расстояний.

Переход от статически заданных массивов к использованию стандартной библиотеки C++ (STL) зачастую является хорошим шагом при оптимизации и улучшении читаемости кода, особенно при работе с комбинаторными задачами.

Память для статического массива будет выделена в стеке, что сильно ограничивает масштабируемость: стековые ресурсы ограничены, и при больших значениях (например, для 10000×10000) программа может выйти за пределы допустимого размера стека.

При использовании `std::vector` или `std::unique_ptr`, возможна динамическая аллокация и освобождение памяти в куче, что позволяет более гибко контролировать использование ресурсов. Фиксированный стековый массив не дает этой возможности и ограничивает возможности управления памятью.

При передаче массива в функции C++ не предоставляет информации о его размерах, поэтому требуется дополнительный код для передачи и проверки размеров. Использование векторов (`std::vector`) или указателей (`std::unique_ptr`) позволяет избежать этой проблемы за счет более гибкого интерфейса.

При доступе к большим статическим массивам, если они не используются полностью, кэш-память будет заполнена ненужными данными. Это может привести к кэш-промахам, что особенно негативно сказывается на производительности при частом доступе к подмножеству элементов массива.

STL-контейнеры и алгоритмы создают чистый, лаконичный и понятный код, который легче понимать и поддерживать.

Переход к STL повышает совместимость кода с другими библиотеками, которые также используют стандартные контейнеры и алгоритмы STL. Это особенно полезно при интеграции различных библиотек для задач параллелизации и работы с графами (в том числе, PPL).

Из минусов использования такого подхода отметим, что в задачах с высокими требованиями к производительности STL-контейнеры могут вводить существенные накладные расходы за счет добавления уровня абстракции. Для исключения замедления работы программы необходим анализ производительности измененного кода.

2.3. *Параллелизация и распределенные вычисления.*

Применение OpenMP.

Технология OpenMP используется для распараллеливания этапа поиска соединения циклов с минимальным весом в функции CycleMerging-Algorithm.

Внутри цикла `while`, который поочередно объединяет циклы, используется параллельный цикл `pragma omp parallel for`, чтобы одновременно оценить возможные комбинации с каждым циклом:

Листинг 1. Использование OpenMP для распараллеливания вычисления стоимости соединения циклов.

```
#pragma omp parallel for shared(minCombinedWeight, bestCombinedCycle,
    bestIndex)
for (size_t i = 0; i < cycles.size(); ++i) {
    vector<int> combinedCycle;
    int combinedWeight = findMinimumCombinedCycle(costMatrix, baseCycle
        , cycles[i], combinedCycle);

    #pragma omp critical
    {
    if (combinedWeight < minCombinedWeight) {
    minCombinedWeight = combinedWeight;
    bestCombinedCycle = combinedCycle;
    bestIndex = i;
    }
    }
}
```

Задача параллелизуется, поскольку оценка стоимости соединений циклов не зависит от других циклов, что позволяет одновременно обрабатывать несколько комбинаций и сократить общее время выполнения.

Применение `pragma omp critical`: так как `minCombinedWeight`, `best-CombinedCycle` и `bestIndex` обновляются, важно избежать условий гонки, для чего блок `omp critical` ограничивает доступ к этим переменным одним потоком.

В результате использования такого подхода ожидается существенное снижение времени вычислений за счет параллельного исполнения, особенно при большом числе циклов.

Использование MPI

MPI используется для распределения тестового набора задач коммивояжера между несколькими процессами. В этом случае MPI позволяет использовать несколько узлов, в

каждом из которых может быть множество потоков, что позволяет реализовать распределенные вычисления.

Для передачи структурированных данных в MPI создается специальный тип `MPI_Datatype` для структуры `Result` с помощью функции `create_mpi_result_type` (листинг 1). Она позволяет передавать результаты вычислений между процессами. Этот тип необходим для передачи данных между узлами, когда каждый узел выполняет свою часть задачи, затем собирает и объединяет результаты:

Листинг 2. Создание пользовательского типа MPI для структуры Result.

```
void create_mpi_result_type(MPI_Datatype& mpi_result_type) {
    int count = 2;
    int block_lengths[2] = { 1, 1 };
    MPI_Datatype types[2] = { MPI_INT, MPI_DOUBLE };
    MPI_Aint offsets[6];

    offsets[0] = offsetof(Result, totalParallelTSPSolutionCost);
    offsets[1] = offsetof(Result, totalParallelTime);

    MPI_Type_create_struct(count, block_lengths, offsets, types, &
       mpi_result_type);
    MPI_Type_commit(&mpi_result_type);
}
```

MPI распределяет данные среди процессов, которые выполняют `solve_tsp_task` с различными участками данных. Использование MPI для решения набора задач коммивояжера позволяет распределить нагрузку на несколько узлов или компьютеров, что значительно ускоряет вычисления при решении крупных задач или больших наборов задач.

Выбор MPI обусловлен тем, что задача требует больших вычислительных мощностей, OpenMP подходит для параллелизации внутри узла, MPI же дает возможность распараллеливания на уровне нескольких узлов.

Внесенные правки позволяют ожидать значительного улучшения производительности при условии, что объем задачи велик, и оправдано распределение нагрузки между узлами.

— *время выполнения*: эта метрика показывает общее время, затраченное программой на выполнение задачи от начала до завершения. В ходе экспериментов фиксировались значения продолжительности выполнения для базовой и оптимизированных версий программы, что позволяет наглядно оценить эффект от улучшений. Измерения проводились на случайно сгенерированных матрицах стоимости для задачи коммивояжера с размерностью от 100 до 1000 с шагом в 100, чтобы показать, как масштабирование задачи влияет на время исполнения.

— *ускорение* (speedup): ускорение является ключевым показателем эффективности оптимизаций и вычисляется как отношение времени выполнения исходного кода к времени выполнения оптимизированного.

— *процент загрузки ЦП*: этот показатель отражает долю времени, в течение которого процессор активно выполняет задачи программы, по сравнению с временем ожидания или бездействия. Высокий процент загрузки ЦП указывает на то, что ресурсы задействованы эффективно, что особенно важно для вычислительно интенсивных задач.

— *использование памяти*: эта метрика показывает объем памяти, потребляемый программой в процессе работы. Оптимизация использования памяти позволяет сократить

нагрузку на систему, избегая излишнего использования ОЗУ, что особенно важно при работе с большими данными.

3. Описание вычислительного эксперимента. Интерес представляют три состояния программы: базовое, оптимизированная версия с использованием стандартной библиотеки шаблонов (STL) и венгерского алгоритма для решения задачи о назначении, а также оптимизированная версия с использованием параллелизма и многопоточности.

Тестирование проводилось в двух вычислительных средах:

— *среда 1 (локальное исполнение).* Персональный компьютер с процессором Intel(R) Core(TM) i5-10210U CPU @ 1.60 GHz (макс. частота 2.11 GHz), 16 ГБ оперативной памяти. В данной среде проводились запуски всех трех версий программы: базовой, оптимизированной и параллельной в однопроцессорной конфигурации.

— *среда 2 (кластерная, распределенное исполнение).* Виртуальные машины, развернутые с использованием QEMU/KVM на отдельных физических узлах. Каждая ВМ имела: Ubuntu 24.04, 4 ГБ ОЗУ, 8 виртуальных процессоров (vCPU) Intel(R) Xeon(R) Silver 4214R @ 2.40GHz, 20 ГБ дискового пространства.

Метрики производительности:

— *время выполнения* — общее время выполнения задачи от начала до завершения. Измерения проводились на случайно сгенерированных матрицах стоимости для задачи коммивояжера размером от 100 до 1000 вершин с шагом 100, что позволяет проанализировать масштабирование по размеру задачи.

— *ускорение* (speedup) — отношение времени выполнения базовой версии к времени выполнения оптимизированной. Данная метрика позволяет оценить прирост производительности вследствие оптимизации и/или распараллеливания.

— *эффективность распараллеливания (по MPI)* — метрика, оценивающая масштабируемость MPI-компоненты программы. Рассчитывается как отношение ускорения к числу MPI-процессов и отражает, насколько эффективно каждая вычислительная единица (MPI-процесс) вносит вклад в общее ускорение.

— *процент загрузки ЦП* — доля времени, в течение которого процессор был задействован в вычислениях. Высокие значения этой метрики свидетельствуют об эффективном использовании вычислительных ресурсов.

— *использование памяти* — объем оперативной памяти, потребляемый программой в процессе выполнения. Оптимизация потребления памяти снижает нагрузку на систему и особенно важна при работе с крупными входными данными.

4. Результаты вычислительного эксперимента.

4.1. *Время выполнения программы.* Рассмотрим, как повлияли изменения в коде на время решения задачи коммивояжера. На рис. 1 отражено время выполнения программы в секундах. Очевидно, переход к решению задачи о назначении венгерским алгоритмом и изменение структуры данных оказали существенное влияние: базовая реализация работает значительно медленнее. Эффект от использования методов параллелизации также положительный, время работы параллельной программы сократилось по сравнению с последовательной реализацией оптимизированного кода.

4.2. *Ускорение.* Рассчитаем коэффициенты ускорения для оптимизированной (S_{opt}) и параллельной (S_{par}) версий относительно базовой, используя формулу:

$$S_{\text{opt}} = \frac{T_{\text{basic}}}{T_{\text{opt}}}, \quad S_{\text{par}} = \frac{T_{\text{basic}}}{T_{\text{par}}},$$

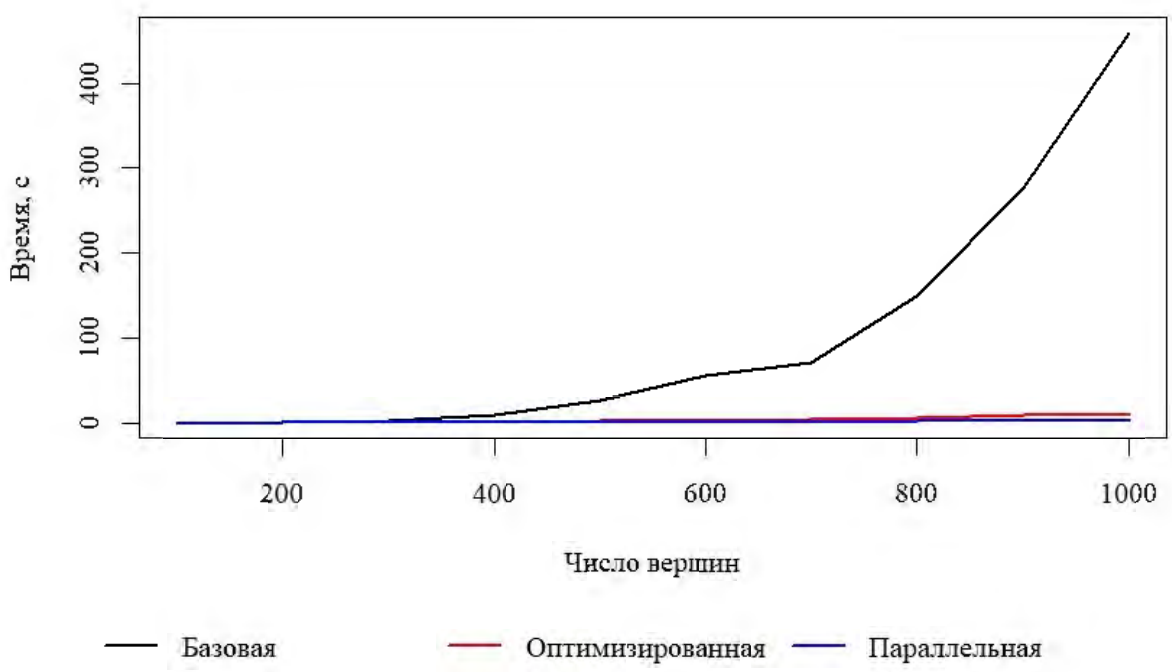


Рис. 1. Время решения одного экземпляра ЗК

где:
— T_{basic} — время выполнения базовой версии,
— T_{opt} — время выполнения оптимизированной версии,
— T_{par} — время выполнения параллельной версии.

На рис. 2 приведены значения ускорения для задач различной размерности (n — число вершин в задаче коммивояжера).

Для оптимизированной версии программы коэффициент ускорения составляет от 0,25 для $n = 100$ до 43,57 для $n = 1000$ (n — число вершин задачи коммивояжера), что показывает заметное сокращение времени работы программы по сравнению с базовой версией. Особенно высокие значения коэффициента ускорения наблюдаются на задачах большой размерности, что объясняется эффективным использованием алгоритмических оптимизаций, позволяющих значительно сократить время выполнения программы при увеличении числа вершин.

Использование методов параллелизации дало значительный положительный эффект: ускорение параллельной реализации по сравнению с последовательной оптимизированной версией составляет от 2,8 до 4,6 раз в зависимости от количества вершин в диапазоне $[100, 1000]$. В сочетании с другими примененными методами оптимизации общее сокращение времени выполнения по сравнению с базовой версией программы превышает сто раз для задач большой размерности.

Это показывает, что использование параллелизации в сочетании с другими методами оптимизации позволяет достичь значительного прироста производительности, что делает подход особенно ценным для практических применений, где требуется быстрое решение сложных вычислительных задач.

4.3. Оценка масштабируемости MPI. В эксперименте использовалась гибридная модель параллелизма (MPI + OpenMP). Число потоков OpenMP было зафиксировано равным 8 на каждый MPI-процесс, что соответствует числу ядер на одном сокете. Масшта-

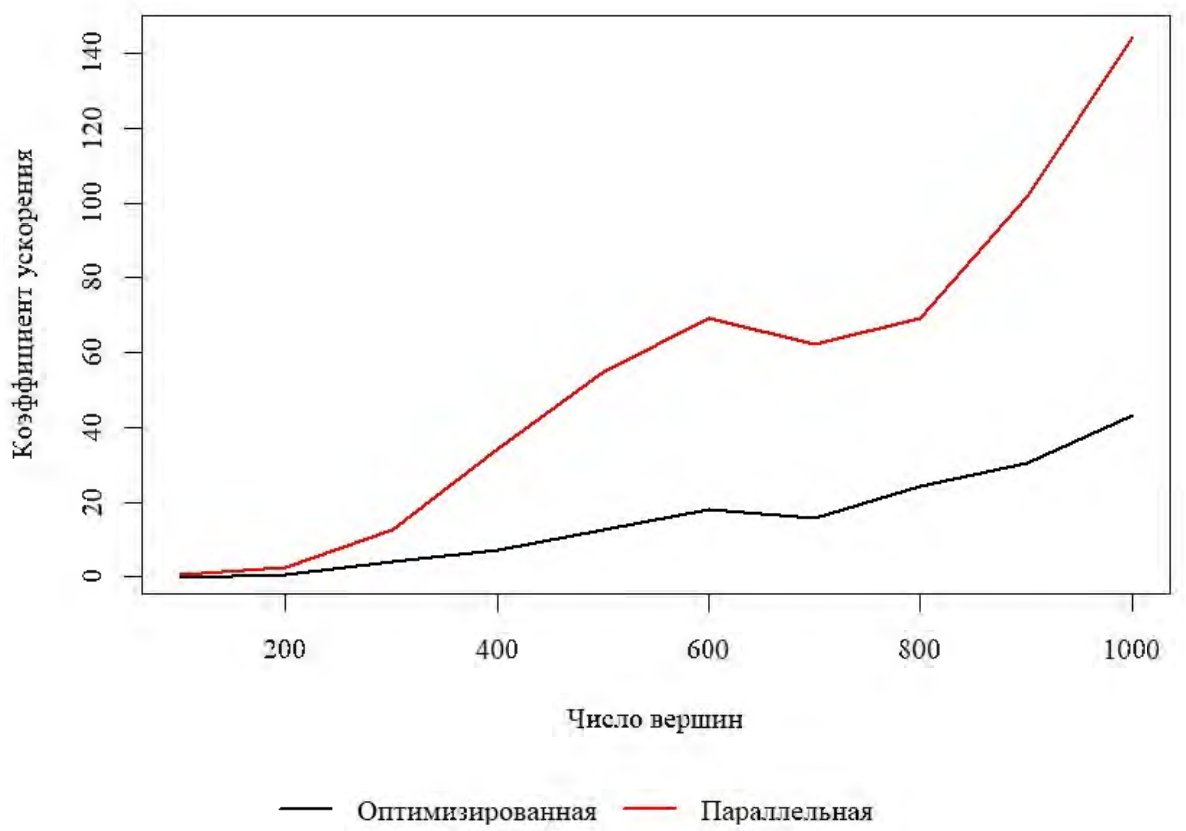


Рис. 2. Коэффициент ускорения по отношению к базовой реализации программы

бирование осуществлялось за счет увеличения числа MPI-процессов: от 8 до 32, с равномерным распределением по доступным узлам (до четырех машин). Таким образом, общее количество задействованных ядер варьировалось от 64 (8 MPI-процессов $\times$ 8 потоков) до 256 (32 $\times$ 8).

Полученные результаты иллюстрируют масштабирование алгоритма при увеличении числа вычислительных узлов, сохраняя фиксированную конфигурацию OpenMP внутри каждого MPI-процесса.

На каждом этапе измерялось общее время решения набора из 100 задач.

С ростом числа MPI-процессов наблюдается значительное снижение времени решения задач. Так, при увеличении числа процессов с 8 до 32 ускорение составляет до 3,6 раз для задач с 1000 вершинами (рис. 3). Это объясняется улучшением распределения нагрузки между вычислительными узлами и снижением времени выполнения параллельных участков кода. Однако при переходе от 24 к 32 процессам прирост производительности становится менее выраженным, что связано с ограничениями по масштабируемости и возрастающими накладными расходами на коммуникации между процессами.

Эффективность распараллеливания представлена на рис. 4 и составляет более 80 % даже при максимальной загрузке, что указывает на хорошую масштабируемость.

4.4. *Загрузка ЦП.* Детальная информация о загрузке центрального процессора и использовании памяти была собрана с помощью *Intel VTune Profiler* [17]. Этот инструмент предоставляет подробные данные о производительности программы, выявляя узкие места

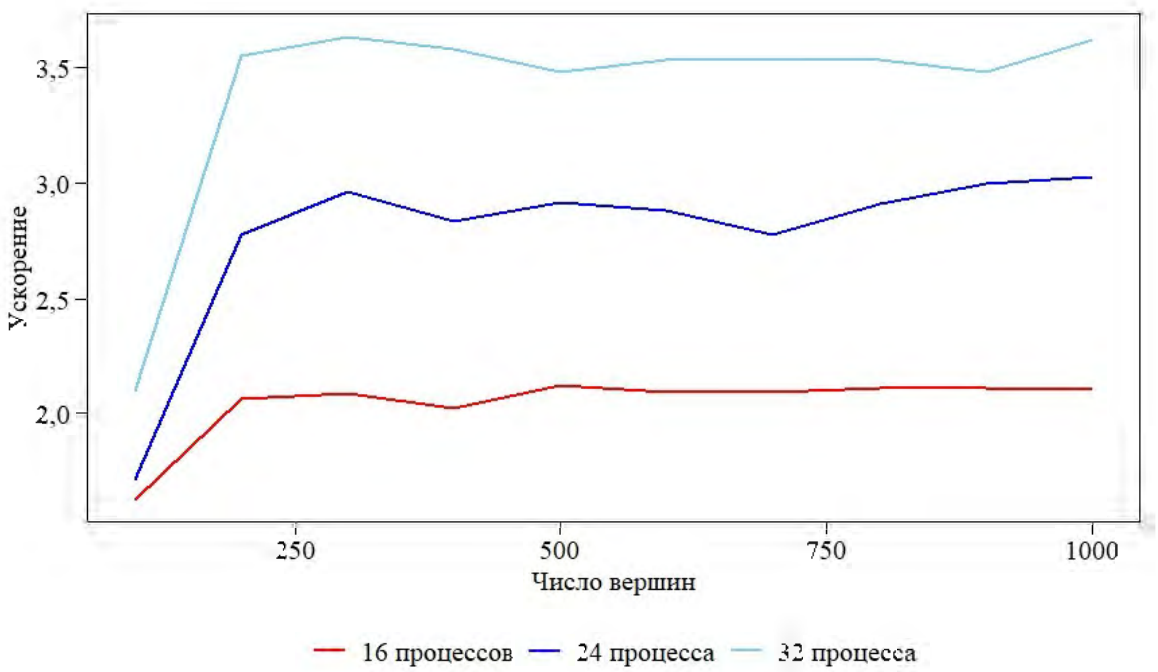


Рис. 3. Ускорение при увеличении числа MPI-процессов

в доступе к памяти и ресурсоемкие участки, а также позволяет анализировать распараллеленный код.

Рис. 5 отражает среднюю загрузку процессора вычислениями приложения для последовательной и параллельной оптимизированных реализаций решения ЗК на 3000 вершин.

Последовательная программа использует один логический процессор из восьми доступных, параллельная реализация загружает 7–8 процессоров и обеспечивает значительное ускорение выполнения, эффективно распределяя нагрузку на вычислительные ресурсы. Такая реализация позволяет достичь более высокой производительности за счет максимального использования доступных процессоров и сокращения времени решения задачи, что особенно важно при решении задач большой размерности.

4.5. *Использование памяти.* В вопросе оптимизации использования памяти особый интерес представляет сравнение базовой и оптимизированной версий программы. В табл. 2 приведены показатели использования памяти и производительности кэша.

Время выполнения. Оптимизированная версия завершает работу за 2,644 секунды, в то время как базовая версия требует 72,505 секунд. Это примерно в 27 раз быстрее.

CPU Time (время, проведенное на CPU) также значительно меньше в оптимизированной версии (2,344 секунды против 66,156 секунд), что говорит о меньшей загрузке процессора благодаря лучшей организации вычислений и обращения к памяти.

Память.

Memory Bound (время, проведенное в ожидании данных из памяти) также снизилось с 15,2 % до 9,2 %, что указывает на более эффективное использование кэша и снижение частоты кэш-промахов.

L1 Bound, L2 Bound, L3 Bound: рост L1 Bound с 2,7 % до 7,7 % указывает на увеличение зависимости программы от данных, хранящихся в кэше L1. Поскольку оптимизация включает использование ссылок и указателей, а также выравнивание данных для улуч-

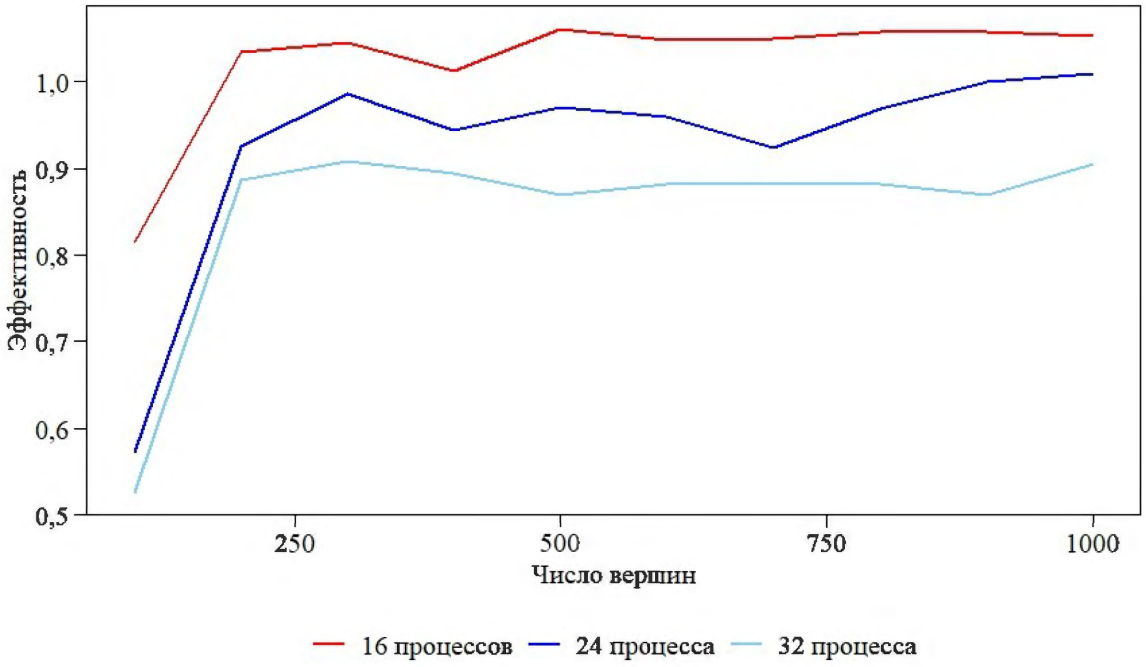
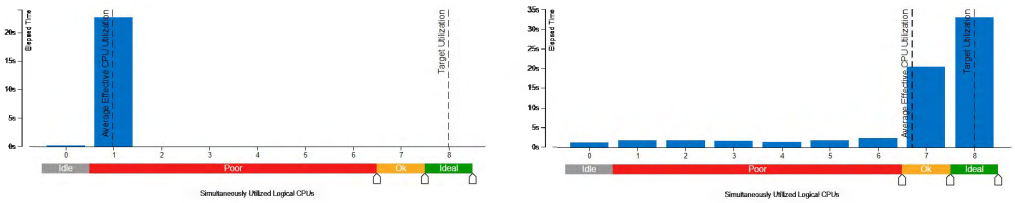


Рис. 4. Эффективность параллельного решения



Последовательная реализация Параллельная реализация

Рис. 5. Эффективность использования ЦП

шения кэш-памяти, это изменение может свидетельствовать о том, что программа теперь эффективнее загружает данные в кэш L1 и быстрее их обрабатывает. Высокий показатель L1 Bound может также указывать на интенсивное использование данных, которые подходят для хранения в L1, что является положительным моментом, так как обращение к L1 кэшу быстрее, чем к более низким уровням памяти. Таким образом, увеличение L1 Bound в данном контексте указывает на то, что оптимизация улучшила использование кэша L1, хотя и добавила небольшие задержки при обращении к данным в этом кэше. На уровнях L2 и L3 наблюдается уменьшение задержек кэша, особенно на L3 уровне, где в оптимизированной версии он снизился до 3,6 % (с 8,8 % в базовой).

DRAM Bound (время, проведенное в ожидании данных из оперативной памяти) увеличилось с 2,7 % до 5,2 %, что говорит о том, что оптимизация снизила интенсивность работы с кэшем, но привела к несколько большей зависимости от данных, загружаемых из оперативной памяти. Это может быть связано с тем, что более эффективная работа с кэшами позволила системе быстрее обрабатывать данные, приводя к увеличению частоты обращения к оперативной памяти по мере исчерпания данных в кэшах. Однако даже с

Таблица 2

Использование памяти базовой и оптимизированной версиями программы

Показатель	Базовая	Оптимизированная
Elapsed Time	72,505 с	2,644 с
CPU Time	66,156 с	2,344 с
Memory Bound	15,2 %	9,2 %
L1 Bound	2,7 %	7,7 %
L2 Bound	2,9 %	0,6 %
L3 Bound	8,8 %	3,6 %
DRAM Bound	2,7 %	5,2 %
Store Bound	0,0 %	0,0 %
Loads	222 623 878 516	4 515 035 447
Stores	42 565 876 938	521 315 639
LLC Miss Count	48 103 367	3 900 273
Average Latency (cycles)	13	9
Total Thread Count	4	4
Paused Time	0 с	0 с

этим увеличением показатель DRAM Bound остается низким (5,2 %), что свидетельствует о хорошей общей эффективности работы с памятью.

Количество операций загрузки и хранения.

Loads (загрузки): оптимизированная версия сделала 4 515 035 447 загрузок по сравнению с 222 623 878 516 в базовой. Это примерно в 49 раз меньше операций, что значительно уменьшает нагрузку на память.

Stores (записи): аналогично, число операций записи также значительным сократилось — примерно в 82 раза.

Кэш-промахи (LLC Miss Count): оптимизированная версия показала заметное снижение кэш-промахов (3 900 273 по сравнению с 48 103 367), что указывает на более эффективное использование кэша в оптимизированной версии и уменьшение необходимости обращаться к более медленным уровням памяти.

Средняя задержка.

Average Latency (cycles): среднее число тактов для задержки при загрузках также сократилось с 13 до 9 циклов. Это говорит о том, что оптимизация привела к более эффективному использованию локальных данных и меньшему количеству обращений к удаленной памяти, что также уменьшает задержки.

Простои CPU (Paused Time): Paused Time равно 0 секунд в обоих случаях, что указывает на отсутствие значительных задержек, не связанных с памятью или вычислениями.

Таким образом, оптимизация привела к значительным улучшениям за счет сокращения количества загрузок и записей, уменьшения кэш-промахов и использования более эффективного доступа к памяти. Эти изменения позволили улучшить использование CPU и значительно сократить время выполнения, что свидетельствует о высоком качестве проведенной оптимизации.

Заключение. Проведенные в рамках данной работы оптимизации продемонстрировали значительное повышение производительности алгоритма решения задачи коммивояжера, обеспечив более эффективное использование вычислительных ресурсов. Реализа-

ция ряда методик, включая оптимизацию функций, эффективное управление памятью и использование технологий параллелизма, таких как MPI и OpenMP, позволили не только сократить время выполнения, но и добиться сбалансированной загрузки процессора и уменьшить потребление оперативной памяти. Проведенные экспериментальные исследования подтвердили, что предложенные улучшения способны успешно масштабироваться на задачи больших размеров, сохраняя высокую точность и надежность вычислений. Данный подход может быть использован и в других задачах комбинаторной оптимизации, где критически важны быстродействие и оптимальное распределение ресурсов, что открывает перспективы для дальнейших исследований и практических приложений в данной области.

В качестве направлений будущих исследований можно отметить возможность интеграции более современных параллельных библиотек или использования GPU-ускорения для вычислительно сложных частей алгоритма, а также сравнение полученных результатов с другими параллельными алгоритмами для приближенного решения задачи коммивояжера.

Список литературы

1. Jarrah A., Bataineh A. S. A., Almomany A. The optimisation of travelling salesman problem based on parallel ant colony algorithm // International Journal of Computer Applications in Technology. 2022. V. 69. N 4. P. 309–321.

2. Rhee Y. Gpu-based parallel ant colony system for traveling salesman problem // Journal of The Korea Society of Computer and Information. 2022. V. 27. N 2. P. 1–8.

3. Wang Z. et al. A fine-grained fast parallel genetic algorithm based on a ternary optical computer for solving traveling salesman problem // The Journal of Supercomputing. 2023. V. 79. N 5. P. 4760–4790.

4. Peng C. Parallel genetic algorithm for travelling salesman problem // In: International conference on automation control, algorithm, and intelligent bionics (ACAIB 2022). 2022. V. 12253, P. 259–267.

5. Alhenawi E. et al. Solving Traveling Salesman Problem Using Parallel River Formation Dynamics Optimization Algorithm on Multi-core Architecture Using Apache Spark // International Journal of Computational Intelligence Systems. 2024. V. 17. N 1. P. 4.

6. Qiao Y. et al. A hybridized parallel bats algorithm for combinatorial problem of traveling salesman // Journal of Intelligent & Fuzzy Systems. 2020. T. 38. N 5. P. 5811–5820.

7. Король З. А. Анкудинов К. А., Королькова Л. Н. Исследование методов оптимизации программного кода для повышения производительности // Инновационные направления развития в образовании, экономике, технике и технологиях. 2023. С. 257–260.

8. Тайк А. М. Лупин С. А., Федяшин Д. А. Использование библиотеки MPI для параллельной реализации алгоритма полного перебора вариантов // Программные продукты и системы. 2023. Т. 36. № 4. С. 607–614.

9. Panyukov A. V., Leonova Y. F. Algorithm for approximate solution of the travelling salesman problem // Optimization Problems and Their Applications (OPTA-2018). 2018. P. 31. (in Russian).

10. Leonova Yu. F. Cycles merging algorithm for an approximate solution of the traveling salesman problem // Abstracts of the XIX All-Russian Conference of Young Scientists on Mathematical Modeling and Information Technologies, Novosibirsk: ICT SB RAS. 2018. P. 28.

11. Panyukov A. V., Leonova Yu. F. Cycle Merging Algorithm for MAX TSP Problems // XVIII International Conference “Mathematical Optimization Theory and Operations Research” (MOTOR 2019), Ekaterinburg, Russia: Publisher “UMC UrFU”. 2019. P. 57.

12. Panyukov A. V., Leonova Yu. F. Cycle merging algorithm for the maximal metric traveling salesman problem // Bulletin of the South Ural State University, V. 10, N 4: a series of Computational Mathematics and Informatics. Chelyabinsk: Publishing House of SUSU. 2021. P. 26–36. (in Russian) DOI: 10.14529/cmse210402.

13. Свидетельство о государственной регистрации программы для ЭВМ № 2021669214 Российская Федерация. Программа для реализации алгоритма соединения циклов для решения задачи коммивояжера : № 2021668239: заявл. 16.11.2021 : опубл. 25.11.2021 / Ю. Ф. Леонова, А. В. Панюков ; заявитель Федеральное государственное автономное образовательное учреждение высшего образования «Южно-Уральский государственный университет». EDN RPNSPO.

14. Гантерот К. Оптимизация программ на C++. Проверенные методы повышения производительности. 1-е изд. М.: Вильямс, 2017. 400 с.

15. Панюков А. В., Телегин В. А. Техника программной реализации потоковых алгоритмов // Вестник Южно-Уральского государственного университета. Серия: Математическое моделирование и программирование. 2008. № 27 (127). С. 78–99.

16. Леонова Ю. Ф. Parallel implementation of the cycle merging algorithm for solving the traveling salesman problem [Электрон. рес.]: `https://github.com/YuliyaLeonova/CycleMergingAlgorithm.git`, last accessed 2025/05/19.

17. Intel VTune Profiler. [Электрон. рес.]: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html/gs.hfjczc`, last accessed 2024/11/11.

18. Мейерс С. Эффективный и современный C++. 42 рекомендации по использованию C++11 и C++14. М.: Вильямс, 2018. 304 с.

19. std::mersenne_twister_engine. [Электрон. рес.]: `https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine.html`, last accessed 2024/11/02.

20. Best Practices in the Parallel Patterns Library. [Электрон. рес.]: `https://learn.microsoft.com/en-us/cpp/parallel/concrt/best-practices-in-the-parallel-patterns-library?view=msvc-170`, last accessed 2024/10/15.

Леонова Юлия Федоровна — аспирант кафедры системного программирования Южно-Уральского государственного университета. Область научных интересов: приближенные алгоритмы решения задач комбинаторной оптимизации.

Leonova Yuliya Fedorovna is postgraduate student of the Department of System Programming of South Ural State University. Research interests: approximate algorithms for solving combinatorial optimisation problems.

Дата поступления — 16.12.2024