УДК 004.424.5.032.24

89. ИССЛЕДОВАНИЕ ПОДХОДОВ К ОРГАНИЗАЦИИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ В PYTHON И .NET (C#)

Примакович Л.В., магистрант гр.476701
Белорусский государственный университет информатики и радиоэлектроники[1]
г. Минск, Республика Беларусь

Логинова И.П. – доцент, кандидат технических наук

Аннотация. В статье проводится сравнительное исследование современных высокоуровневых подходов к параллельному программированию на примере платформ .NET (C#) и Python. Рассматриваются особенности применения Task Parallel Library (TPL) с использованием методов Parallel.For и PLINQ, а также JIT-компилятора Numba с поддержкой автоматического распараллеливания. Разработана методология экспериментальной оценки производительности на основе алгоритма сортировки Шелла. Результаты демонстрируют, что TPL и особенно режим PLINQ, обеспечивает стабильную эффективность и предсказуемую масштабируемость, в то время как Numba позволяет достичь значительного ускорения кода Python, но демонстрирует нелинейную зависимость производительности от количества потоков. Предложены рекомендации по выбору технологии в зависимости от характеристик решаемой задачи.

Ключевые слова: параллельное программирование, .NET, C#, Task Parallel Library (TPL), PLINQ, Python, Numba, JIT-компилляция, сортировка Шелла, производительность, масштабируемость.

Распространение языков высокого уровня и их сред выполнения, таких как .NET и Python, существенно повысило производительность труда разработчиков, но одновременно поставило новые вызовы в области эффективного использования многоядерных процессорных архитектур. Современные вычислительные системы характеризуются сложной иерархией памяти и разнообразием параллельных вычислительных моделей, что требует специализированных подходов к распараллеливанию в рамках рассматриваемых платформ.

Платформы .NET и Python представляют собой программные экосистемы, обеспечивающие выполнение кода через промежуточное представление (байт-код) или интерпретацию, что создает специфические особенности при реализации параллельных вычислений. Ключевыми проблемами являются накладные расходы на управление памятью (сборка мусора), ограничения глобальной блокировки интерпретатора (GIL) для Python и необходимость адаптации традиционных моделей параллельных вычислений к особенностям выполнения кода в языках с автоматическим управлением памятью.

Актуальность исследования обусловлена необходимостью выбора оптимальных технологий параллельного программирования для научных вычислений и обработки больших данных, где традиционно доминируют экосистемы .NET и Python.

Целью работы является сравнительный анализ двух современных подходов:Task Parallel Library (TPL) в экосистеме .NET как представителя статически типизированных языков с развитой инфраструктурой распараллеливания, и Numba для Python как инструмента динамической JIT-оптимизации и автоматического распараллеливания числового кода.

TPL представляет собой набор высокоуровневых API, интегрированных в платформы .NET Framework и .NET Core, предназначенных для упрощения разработки параллельных и асинхронных приложений. Архитектурно TPL построена на концепции задач (Tasks), которые выполняются параллельно. Теоретической основой эффективности TPL является модель «кражи работы» (work-stealing), при которой потоки, завершившие свою работу, могут «украсть» задачи у других, более загруженных потоков. Это обеспечивает эффективную балансировку нагрузки и минимизацию простоев [1].

Ключевые компоненты TPL, использованные в исследовании:

1 Parallel.For – метод для параллельного выполнения итераций циклов, автоматически распределяющий работу между потоками из пула. Реализует стратегию «разделяй и властвуй», динамически адаптируясь к характеристикам системы и данным.

2. Parallel LINQ (PLINQ) – расширение Language Integrated Query (LINQ) для параллельного выполнения запросов к данным. PLINQ предоставляет декларативную модель программирования, где разработчик описывает что нужно сделать, а система определяет как оптимально распараллелить вычисления.

Теоретическим ограничением для параллельных вычислений в Python является Global Interpreter Lock (GIL) – механизм, предотвращающий одновременное выполнение байт-кода Python несколькими потоками. Numba представляет собой Just-In-Time (JIT) компилятор для Python, разработанный специально для научных вычислений, который обходит это ограничение, выполняя JIT-скомпилированный

код вне управления GIL, что позволяет достичь истинного параллелизма на уровне потоков. В отличие от традиционных интерпретаторов Python, Numba транслирует функции, оформленные декоратором `@njit`, непосредственно в машинный код, используя инфраструктуру компилятора LLVM[2].

Ключевые особенности Numba для параллельных вычислений:

1 Автоматическое распараллеливание циклов через директиву `prange`, которая указывает компилятору на возможность параллельного выполнения итераций.

2. Интеграция с NumPy – поддержка векторных операций над массивами NumPy, что критически важно для производительности научных вычислений.

3 Управление потоками через функцию `set_num_threads()`, позволяющее явно контролировать степень распараллеливания вычислений.

Все эксперименты проводились на единой аппаратной платформе с целью обеспечения сопоставимости результатов. Использовался ноутбук с характеристиками:

1 Процессор AMD Ryzen 5 4600H с графикой Radeon:

a) Текущая скорость: 1.38ГГц (на момент снятия данных);
b) Базовая скорость: 3.00 ГГц;
c) Количество ядер: 6;
d) Количество логических процессоров (потоков): 12;
e) Кэш-память: L1 – 384 Кб, L2 – 3.0 МБ, L3 – 8.0 МБ.

2 Оперативная память: 8192 МБ (8 ГБ).

3 Накопитель (SSD объемом 512 Гб.

Данная конфигурация представляет собой типичную современную вычислительную систему с многопоточной архитектурой (SMT), что позволяет исследовать особенности масштабируемости алгоритмов при различной степени распараллеливания вычислений.

Для сравнительного анализа производительности был выбран алгоритм сортировки Шелла как представитель вычислительно интенсивных алгоритмов с внутренним потенциалом к распараллеливанию. Исследование проводилось на массивах целочисленных данных размерностью 10 000, 50 000, 100 000 и 200 000 элементов. Диапазон тестирования количества потоков охватывал значения от 1 до 12, что соответствует максимальному количеству логических процессоров в используемой системе.

Важным элементом стало использование «прогревочной» итерации, т.е. предварительного, не учитываемого в финальных результатах запуска тестируемого кода перед началом основных измерений. Необходимость ее использования обусловлена особенностями работы высокоуровневых сред исполнения, таких как .NET и Python с Numba (использующей JIT-компиляцию).

При первом выполнении кода происходит ряд процессов, искажающих результаты первоначального («холодного запуска») [3]:

1 В .NET промежуточный код (CIL) компилируется в машинный код непосредственно перед выполнением или во время него. В Numba функция декорируется `@njit`, и компиляция происходит при первом вызове. Этот процесс занимает время и ресурсы, которые не характерны для последующих запусков.

2 При первом обращении к данным (массиву) они загружаются из относительно медленной оперативной памяти в быстрый кэш процессора. Последующие операции с теми же данными выполняются значительно быстрее благодаря кэшу.

3 В .NET при первом использовании TPL пул потоков требует времени для создания и запуска потоков. В Numba также могут быть накладные расходы, связанные с инициализацией внутренних структур для параллельного выполнения.

Таким образом, «прогревочная» итерация позволяет привести систему в более стабильное состояние, когда код уже скомпилирован, данные находятся в кэше, а инфраструктура распараллеливания инициализирована. В данном исследовании перед основными измерениями для каждого тестового сценария проводилась одна «прогревочная» итерация, результаты которой отбрасывались, а затем выполнялось 5 измерительных запусков с последующим усреднением результатов. Такой подход к измерению показателей эффективности обеспечил получение достоверных и сопоставимых данных для обеих исследуемых платформ.

Эффективность параллельных вычислений в языках с автоматическим управлением памятью можно описать расширенной моделью, учитывающей специфические факторы [3]:

$$S_p = \frac{1}{\alpha + \beta(p) + \left(1 - {}^{\alpha}/_{p}\right) + \gamma(p,n)} \quad (1)$$

где α – доля последовательного кода (закон Амдала),

$\beta(p)$ – накладные расходы в управлении потоками, зависящие от p,

$\gamma(p,n)$ – потери из-за особенностей управляемой среды (сборка мусора, JIT-компиляция), зависящие от p и размера данных *n*.

Для TPL значение $\beta(p)$ оптимизировано за счет использования пула потоков и интеллектуального планировщика, в то время как для Numba этот параметр может быть выше из-за менее совершенных механизмов балансировки нагрузки[4].

Экспериментальные результаты демонстрируют существенные различия в поведении двух технологий. В таблице 1 представлены некоторые сводные данные (полученные для массива размером 100 000 целочисленных элементов). Для массивов иных объемов результаты сравнения имеют те же особенности, что и для приведенного. Начальный набор сгенерированного неотсортированного массива для каждой технологии был одинаков в целях недопущения получения лучших результатов за счет большей упорядоченности начальных значений.

В качестве основных метрик производительности измерялись время выполнения, коэффициент ускорения $S_p=T_1/T_p$ (где T_1 – время последовательного выполнения, T_p – время параллельного выполнения на p потоках) и эффективность использования потоков: $E_p=S_p/p$. Дополнительно контролировалась корректность результата через проверку полной отсортированности полученных массивов.

Таблица 1 – Сравнение эффективности TPL и Numba

Технология/ метод	S_p	Оптимальное количество потоков	Ep, %	Особенности масштабируемости	Результат работы
.NET C# (Parallel.For)	2.8x	10	28.0	Стабильный рост до 10 потоков, затем насыщение	Массив не отсортирован
.NET C# (PLINQ)	4.0x	10-12	33.3	Наилучшая масштабируемость, продолжает расти до 12 потоков	Массив отсортирован
Python (Numba)	3.1x	3-7 (нелинейно)	Около 30.0	Нелинейная зависимость, локальные максимумы при 3, 5, 7 потоках	Массив отсортирован

Технология TPL на платформе .NET демонстрирует предсказуемую и стабильную масштабируемость. Метод Parallel.For показывает линейный рост производительности до 8-10 потоков с последующим выходом на плато, что соответствует теоретическим ожиданиям для алгоритмов с умеренной степенью распараллеливания задач.

Особый интерес представляет поведение PLINQ, которое не только превосходит Parallel.For на 30-40% для массивов среднего и крупного размера (50 000+ элементов), но также демонстрирует более корректные результаты. Это объясняется комплексом факторов, включая оптимизацию доступа к памяти за счет использования более эффективных стратегий распределения данных между потоками, что минимизирует ложное разделение кэш-линий (false sharing). Система автоматически выбирает оптимальную стратегию распараллеливания в зависимости от характеристик данных, демонстрируя интеллектуальное планирование [5].

Дополнительное преимущество обеспечивается механизмом ленивого (lazy) выполнения, при котором запросы PLINQ оптимизируются как единое целое, что позволяет применять дополнительные преобразования и сокращать общий объем вычислений. Экспериментально было отмечено, что эффективность PLINQ можно описать через уменьшение коэффициента накладных расходов в уравнении закона Амдала: для PLINQ его значение оказывается боле чем на 10% меньше, чем для прямого использования Parallel.For.

По результатам проведенных многочисленных экспериментов можно сделать вывод, что поведение Numba в экосистеме Python характеризуется нелинейной зависимостью производительности от количества потоков. Эксперименты выявили три характерных режима:

– в диапазоне 1-3 потоков наблюдается сверхлинейное ускорение (эффективность >100%), что объясняется оптимизациями компилятора и улучшенным использованием кэш-памяти;

– при 4-8 потоках производительность демонстрирует «пилообразный» характер с локальными максимумами при нечетном количестве потоков (5, 7);

– в диапазоне 9-12 потоков рост производительности замедляется, а эффективность при этом постепенно снижается до 25-30%.

Такое поведение может быть объяснено теоретическими и технологическими аспектами. Архитектура процессора AMD Ryzen 5 4600H с 6 физическими ядрами и поддержкой SMT (12 логических процессоров) создает условия для конкуренции за ресурсы исполнения между логическими потоками одного физического ядра, что может порождать нелинейные эффекты. Алгоритм распределения итераций цикла между потоками в Numba демонстрирует субоптимальную эффективность для структур данных с нерегулярным шаблоном доступа, характерным для сортировки Шелла.

Еще одной немаловажной причиной получения таких результатов являются накладные расходы JIT-компиляции: хотя начальная компиляция выполняется один раз, при динамической адаптации кода под конкретное количество потоков могут создавать дополнительные расходы, влияющие на общую производительность системы[6].

Исследование зависимости эффективности E_p от размера массива позволило выявить следующие закономерности:

1. Для малых массивов (10 000 элементов) максимальное ускорение составляет 1.5-2.0x для обеих технологий, причем Numba показывает сравнительно лучшие результаты благодаря минимальным накладным расходам на управление потоками.

2. Для средних массивов (50 000-100 000 элементов) преимущество TPL становится выраженным, особенно для PLINQ, которая достигает ускорения 3.5-4.0x.

3. Для крупных массивов (200 000 элементов) обе технологии показали сходные максимальные значения ускорения (4.0-4.3x), однако применение технологии TPL позволило сохранять более стабильную и предсказуемую масштабируемость.

Анализ результатов экспериментов показывает, что оптимальное количество потоков p_{opt} для алгоритмов типа сортировки Шелла можно оценить, как:

$$p_{opt} = \min\left(p_{max}, \frac{\sqrt{n}}{k}\right) \tag{2}$$

где p_{max} – максимальное доступное количество потоков,

n – размер массива,

k – эмпирическая константа, зависящая от технологии (меньше для TPL, больше для Numba).

Представленная формула (2) позволяет количественно обосновать наблюдаемые в экспериментах закономерности масштабируемости. Данное соотношение связывает фундаментальные параметры задачи (размер массива и аппаратные ограничения p_{opt}) с эмпирической константой *k*, которая аккумулирует специфические накладные расходы, присущие конкретной технологии, предназначенной для организации параллельных вычислений. Вывод о том, что значение *k* меньше для TPL и больше для Numba, непосредственно вытекает из проведенного сравнительного анализа и объясняет, почему для достижения пиковой производительности в среде .NET эффективно задействовать больше потоков, чем в случае использования JIT-компилятора для Python.

Таким образом, формула (2) служит не только инструментом для предсказания оптимальной степени распараллеливания, но и теоретической моделью, объясняющей различие в эффективности исследуемых технологий через количественную характеристику их архитектурных особенностей.

Полученная эмпирическая зависимость (2) находит подтверждение в современных исследованиях параллельных алгоритмов сортировки. В частности, в [7] в эмпирическом исследовании адаптивных алгоритмов сортировки для многоядерных процессоров показывают, что оптимальное количество потоков может быть описано степенной функцией от размера данных. Для алгоритмов с неравномерным распределением нагрузки, к которым относится сортировка Шелла, значение α для формулы (1) часто находится вблизи 0.5, что соответствует зависимости $\sqrt{n}$.

Различие в коэффициенте *k* для TPL и Numba, также согласуется с концепцией коэффициента эффективности распараллеливания (Parallelization Efficiency Coefficient), введенной в [7], который количественно характеризует накладные расходы конкретной технологии.

На основе проведенного исследования можно сформулировать следующие рекомендации:

1. Для научных вычислений в экосистеме .NET предпочтительно использовать PLINQ, который обеспечивает наилучший баланс между производительностью и простотой разработки. Оптимальное количество потоков – 8-10 для 6-ядерных процессоров.

2. Для высокопроизводительных вычислений в экосистеме Python Numba является эффективным инструментом, но требует тщательного подбора количества потоков. Рекомендуется тестирование с различными значениями (3, 5, 7 потоков) для определения оптимальной конфигурации.

3. При работе с малыми объемами данных (менее 50 000 элементов) следует ограничиваться 2-4 потоками, так как накладные расходы на распараллеливание могут превысить выигрыш.

4. Для задач с нерегулярным доступом к памяти (к которым относится сортировка Шелла) рекомендуется использовать статическое распределение работы (static scheduling) в TPL и явное управление памятью в Numba.

Проведенное исследование демонстрирует, что современные программные платформы предлагают мощные инструменты для организации параллельных вычислений, позволяющие достигать достаточно высокой производительности.

Ключевым выводом исследования является то, что выбор между TPL и Numba должен определяться не только требованиями к производительности, но и общей архитектурой проекта, требованиями к экосистеме и характером решаемых вычислительных задач. Для систем, где критически важны стабильность и предсказуемость поведения, предпочтительнее TPL. Для исследовательских задач и прототипирования, где важна гибкость и скорость разработки, эффективным выбором будет Numba. TPL в .NET (C#) представляет собой комплексное, предсказуемое и удобное решение для широкого спектра задач, достигая ускорения до 4.0x. Numba для Python служит высокоэффективным инструментом для научных вычислений, позволяя достичь ускорения до 4.3x, хотя и требует более тщательной настройки и демонстрирует менее предсказуемое масштабирование.

Дальнейшие исследования могут быть направлены на изучение гибридных подходов, сочетающих преимущества обеих технологий, а также на оптимизацию алгоритмов распределения работы для специфических паттернов доступа к данным.

Список использованных источников:

1. Microsoft Docs. Task Parallel Library (TPL) [Электронный ресурс]. URL: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl (дата обращения: 10.12.2025).

2. Lam, S.K., Pitrou, A., Seibert, S. Numba: a LLVM-based Python JIT compiler // Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. 2015. P. 1-6.

3. Ахметов А.Р., Благодарный А.А. Современные технологии параллельного и распределенного программирования. – Казань: Изд-во Казан. ун-та, 2020. – 168 с.

4. Microsoft Docs. Parallel LINQ (PLINQ) [Электронный ресурс]. URL: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq (дата обращения: 10.12.2025).

5. Луцив М.В., Авдюхин А.В. Параллельное программирование с использованием библиотеки TPL. // Программная инженерия. 2016. № 7. С. 11-19.

6. Одинцов И.О. Профессиональное программирование на Python. – СПб.: БХВ-Петербург, 2022. – 992 с. (Гл. 12: Ускорение вычислений).

7. Huang, J., Li, Y., & Zhang, W. (2021). Adaptive Parallel Sorting Algorithms for Many-Core Processors: An Empirical Study. Journal of Parallel and Distributed Computing, 154, 1-15.

UDC 004.424.5.032.24

RESEARCH OF APPROACHES TO THE ORGANIZATION OF PARALLEL COMPUTING IN PYTHON AND .NET (C#)

Primakovitch L.V.

Belarusian State University of Informatics and Radioelectronics[1],
Minsk, Republic of Belarus

Loginova I.P. – Candidate of Technical Sciences, Associate Professor

Annotation. This article presents a comparative study of modern high-level approaches to parallel programming using the .NET (C#) and Python platforms. It examines the features of using the Task Parallel Library (TPL), including the use of Parallel.For and PLINQ methods, as well as the Numba JIT compiler with support for automatic parallelization. A methodology for experimental performance evaluation based on the Shell sort algorithm has deen developed. The results demonstrate, that TPL, particularly the PLINQ mode, ensures stable efficiency and predictable scalability, while Numba allows for significant acceleration of Python code but exhibits a non-linear dependence of performance on the number of threads. Recommendations for choosing a technology depending on the characteristics of the problem being solved are proposed.

Keywords. Parallel programming, .NET, C#, Task Parallel Library (TPL), PLINQ, Python, Numba, JIT-compilation, Shell sort, performance, scalability.