Санкт-Петербургский государственный университет

Кафедра системного программирования

Сравнение производительности подсчёта треугольников: GraphBLAS vs Apache Spark

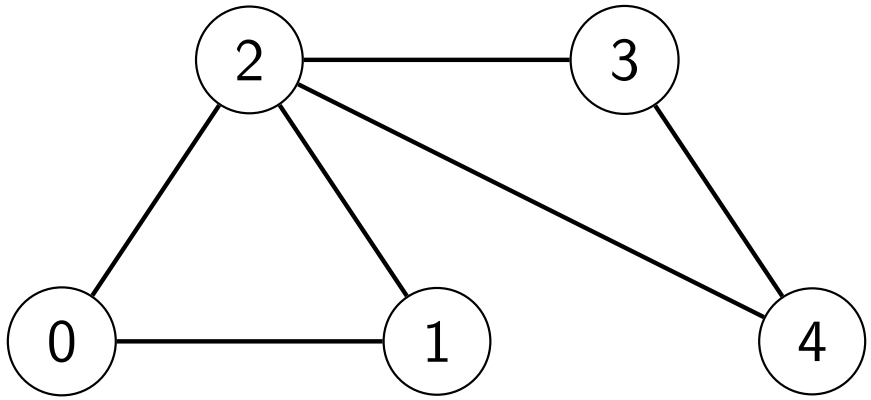
Команда 5: Аверин Павел, Кузнецов Арсений, Якшигулов Вадим

Санкт-Петербург
2025

Постановка задачи подсчёта треугольников

Для неориентированного графа $G = (V, E)$, без петель и кратных рёбер, нужно найти количество треугольников.

Треугольник — тройка вершин, попарно соединённых рёбрами.



Например, в данном графе два треугольника: $(0, 1, 2)$ и $(2, 3, 4)$

Прикладное значение

Анализ социальных сетей:

- Подсчёт треугольников используется для вычисления **коэффициента кластеризации**
- Метрика показывает, насколько узлы склонны образовывать плотные группы
- *Twitter* и *Facebook* используют для анализа формирования сообществ
- Оценка “эффекта эхо-камеры”

Другие применения:

- Обнаружение спама и ботов
- Анализ биологических сетей
- Рекомендательные системы

Алгоритмы подсчёта треугольников

Пусть A — матрица смежности графа, L — нижнетреугольная часть A.

Базовый алгоритм:

$$\frac{\text{trace}(A^3)}{6}$$

Алгоритм Burkhard:

$$\frac{\sum(A^2 \odot A)}{6}$$

где $\odot$ — поэлементное умножение.

Алгоритм Sandia:

$$\sum((L \times L) \odot L)$$

где $\times$ — матричное умножение.

Burkhard: GraphBLAS

```
GrB_mxm(C, graph->A, NULL,
        GxB_PLUS_TIMES_UINT32,
        graph->A, graph->A, d);
GrB_reduce(&ntri, NULL,
           GxB_PLUS_INT64_MONOID, C, NULL);
ntri /= 6;
```

- `GrB_mxm(C, A, ...)` — $C\langle A\rangle = A^2$ (маскированное умножение)
- `GrB_reduce` — суммирование элементов матрицы

Burkhard: Spark — код

```
val matrixA = graph.edges                                    (1)
  .map(e => ((e.srcId, e.dstId), 1L))

val firstTerm = matrixA                                      (2)
  .map { case ((i,k),_) => (k, i) }
val secondTerm = matrixA                                     (3)
  .map { case ((k,j),_) => (k, j) }

val matrixA2 = firstTerm.join(secondTerm)                    (4)
  .map { case (k,(i,j)) => ((i,j), 1L) }
  .reduceByKey(_ + _)

val matrixC = matrixA2.join(matrixA)                         (5)

val totalSum = matrixC                                       (6)
  .map { case (_,(count,_)) => count }
  .reduce(_ + _)
```

Burkhard: Spark — эквивалентность

1. `matrixA` — представление A как `RDD[((i,j), 1)]`
2. `(i,k) → (k,i)` — переиндексация (ключ = средняя вершина)
3. `(k,j) → (k,j)` — переиндексация (ключ = средняя вершина)
4. `join + reduceByKey` — умножение: $(A^2)_{ij} = \sum_k A_{ik} \cdot A_{kj}$
5. `join(matrixA)` — маскирование (оставляем только рёбра из A)
6. `reduce` — суммирование всех элементов

Sandia: GraphBLAS

```
GxB_select(L, NULL, NULL,
           GxB_TRIL, graph->A, Thunk, NULL);
GrB_mxm(C, L, NULL,
        GxB_PLUS_TIMES_UINT32, L, L, d);
GrB_reduce(&ntri, NULL,
           GxB_PLUS_INT64_MONOID, C, NULL);
```

- `GxB_select(GxB_TRIL)` — нижний треугольник: $L_{ij} = A_{ij}$ если $i > j$
- `GrB_mxm(C, L, ...)` — $C\langle L \rangle = L^2$ (маскированное умножение)
- `GrB_reduce` — суммирование всех элементов матрицы C

Sandia: Spark — код

```
val directedEdges = graph.edges.flatMap { e =>                    (1)
  if (e.srcId < e.dstId)
    Some(((e.srcId, e.dstId), 1))
  else if (e.dstId < e.srcId)
    Some(((e.dstId, e.srcId), 1))
  else None
}.reduceByKey(_ + _)

val triangles = directedEdges                                     (2)
  .map { case ((s, d), _) => (d, s) }
  .join(directedEdges.map { case ((s, d), _) => (s, d)})
  .map { case (m, (st, e)) =>                                     (3)
    if (st < e) ((st, e), 1) else ((e, st), 1) }
  .filter { case ((st, e),_) => st < e }
  .join(directedEdges)                                            (4)
  .count()                                                        (5)
```

Sandia: Spark — эквивалентность

1. `flatMap` — построение L: оставляем (i, j) только если $i < j$
2. Первый `join` — умножение: находим пути (i, k, j) в L
3. `map + filter` — нормализация: гарантируем $i < j$
4. Второй `join` — маскирование: проверяем $(i, j) \in L$
5. `count()` — подсчёт треугольников

Итого: $\sum_{i<j} (L^2)_{ij} \cdot L_{ij}$

Конфигурация оборудования

Характеристики вычислительной машины:

- **Процессор:** Apple M3 Pro (12 ядер)
  - 6 производительных ядер
  - 6 энергоэффективных ядер
- **RAM:** 36 GB (unified memory)
- **GPU:** Apple M3 GPU (18 ядер)
- **Кэш:**
  - L1: 256 KB
  - L2: 36 MB (32 MB производительные ядра, 4 MB энергоэффективные)
  - L3: 24 MB
- **ОС:** macOS Sonoma 14.6.1

Оркестрация:

- Kubernetes для управления задачами
- Docker-контейнеры для изоляции

Программное обеспечение

- **GraphBLAS:** SuiteSparse:GraphBLAS 10.0.3
- **Apache Spark:** 3.5.5
- **Scala:** 2.12
- **Kubernetes:** 1.32.6
- **Docker:** 28.3.3

Набор данных

Источник: Stanford Large Network Dataset Collection (SNAP)

Датасет	Вершины	Рёбра	Тип
loc-brightkite	58k	214k	Геолокация
soc-Epinions1	75k	508k	Соцсеть
roadNet-PA	1088k	1.5M	Дорожная сеть
soc-LiveJournal1	4.8M	68.9M	Соцсеть

Обоснование выбора:

- Прогрессия размеров: 214k $\rightarrow$ 508k $\rightarrow$ 1.5M $\rightarrow$ 68.9M рёбер
- Разные типы графов: социальные сети, инфраструктура
- Разная плотность: RoadNet разреженный, остальные плотные

Выбор конфигураций

GraphBLAS: 6 конфигураций

- 6cpu/24Gi
- 3cpu/12Gi
- 2cpu/4Gi
- Алгоритмы: sandia, burkhard

Spark: 9 конфигураций

- 1×6cpu/24Gi — один большой воркер
- 2×3cpu/12Gi — средняя распределённость
- 3×2cpu/8Gi — много мелких воркеров
- Алгоритмы: sandia, burkhard, graphx
- Driver: 1cpu/2Gi для всех

Постановка эксперимента

Предположения:

- Sandia будет быстрее Burkhard на всех платформах
- GraphX будет быстрее самописных алгоритмов на Spark
- GraphBLAS будет быстрее Spark во всех тестах
- GraphBLAS будет падать по OOM на больших графах
- Spark на тех же графах выполнится из-за вытеснения на SSD
- Spark: умеренный параллелизм быстрее всего на мелких графах

Методология:

- 20 запусков каждой конфигурации
- Доверительные интервалы 95%

Ограничения исследования

Важно понимать:

- Тестирование проводится на **одной физической машине**
- Spark запускается с несколькими воркерами через Kubernetes
- Мы измеряем **оверхед распределённой архитектуры** в условиях single-node
- GraphBLAS оптимизирован для single-node, Spark — для кластеров

Цель: Определить, при каких размерах графов и конфигурациях оверхед Spark становится неприемлемым на одной машине.

Результаты экспериментов

Таблица: GraphBLAS: среднее время выполнения (секунды)

Датасет	**Sandia** 6/24	**Sandia** 3/12	**Sandia** 2/4	**Burkhard** 6/24	**Burkhard** 3/12	**Burkhard** 2/4
Brightkite	✓ 0.11	✓ 0.12	✓ 0.14	✓ 0.13	✓ 0.13	✓ 0.22
Epinions	✓ 0.12	✓ 0.13	✓ 0.18	✓ 0.17	✓ 0.27	✓ 0.33
RoadNet-PA	✓ 0.18	✓ 0.32	✓ 0.41	✓ 0.25	✓ 0.26	✓ 0.49
LiveJournal	✓ 7.5	✓ 15.5	✓ 28.1	✓ 41.9	✓ 87.0	✗ OOM

- Нормальность распределения данных проверена тестами Шапиро-Уилка и Д'Агостино ($p > 0.05$)

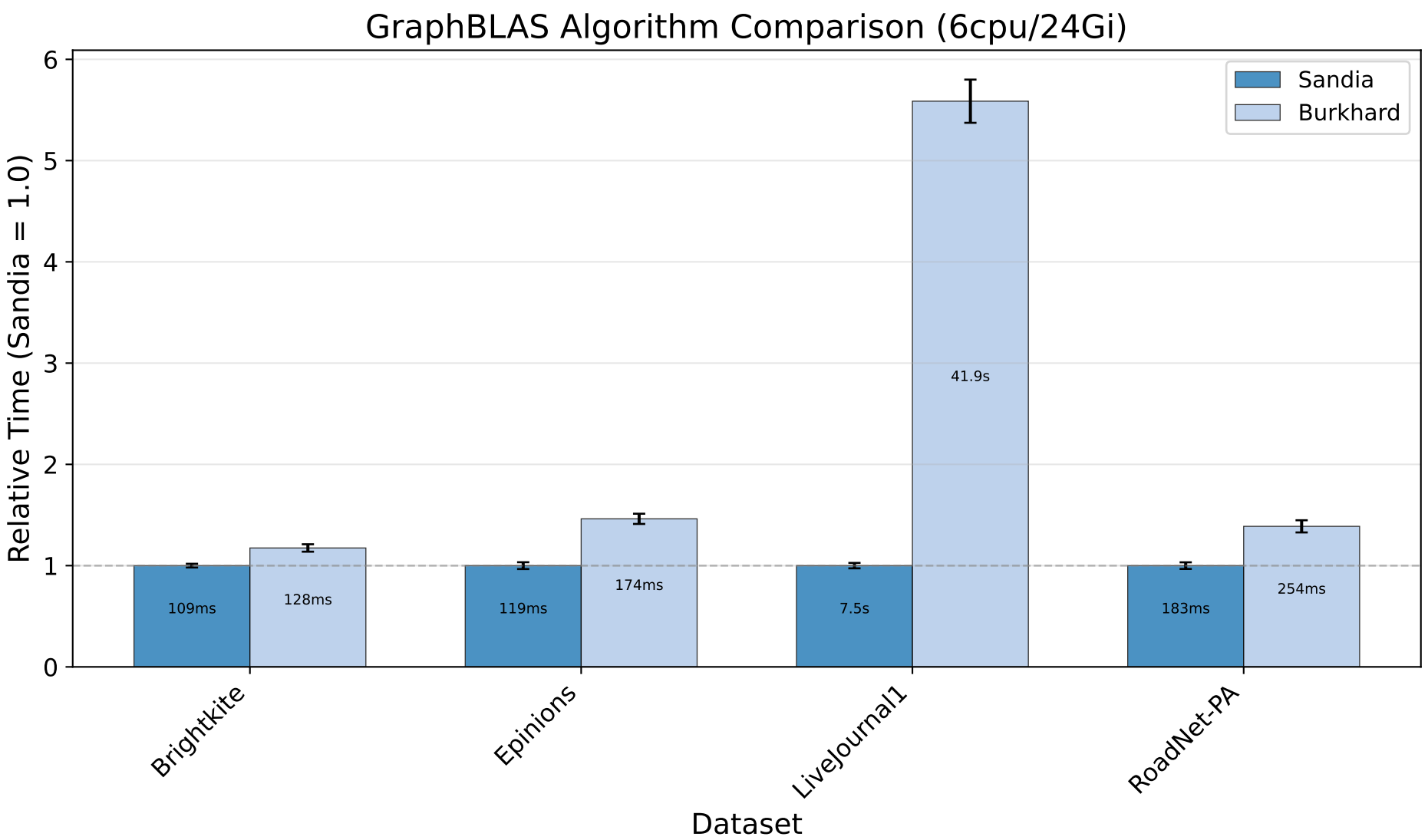
Результаты экспериментов (продолжение)

Таблица: Spark: среднее время выполнения (секунды)

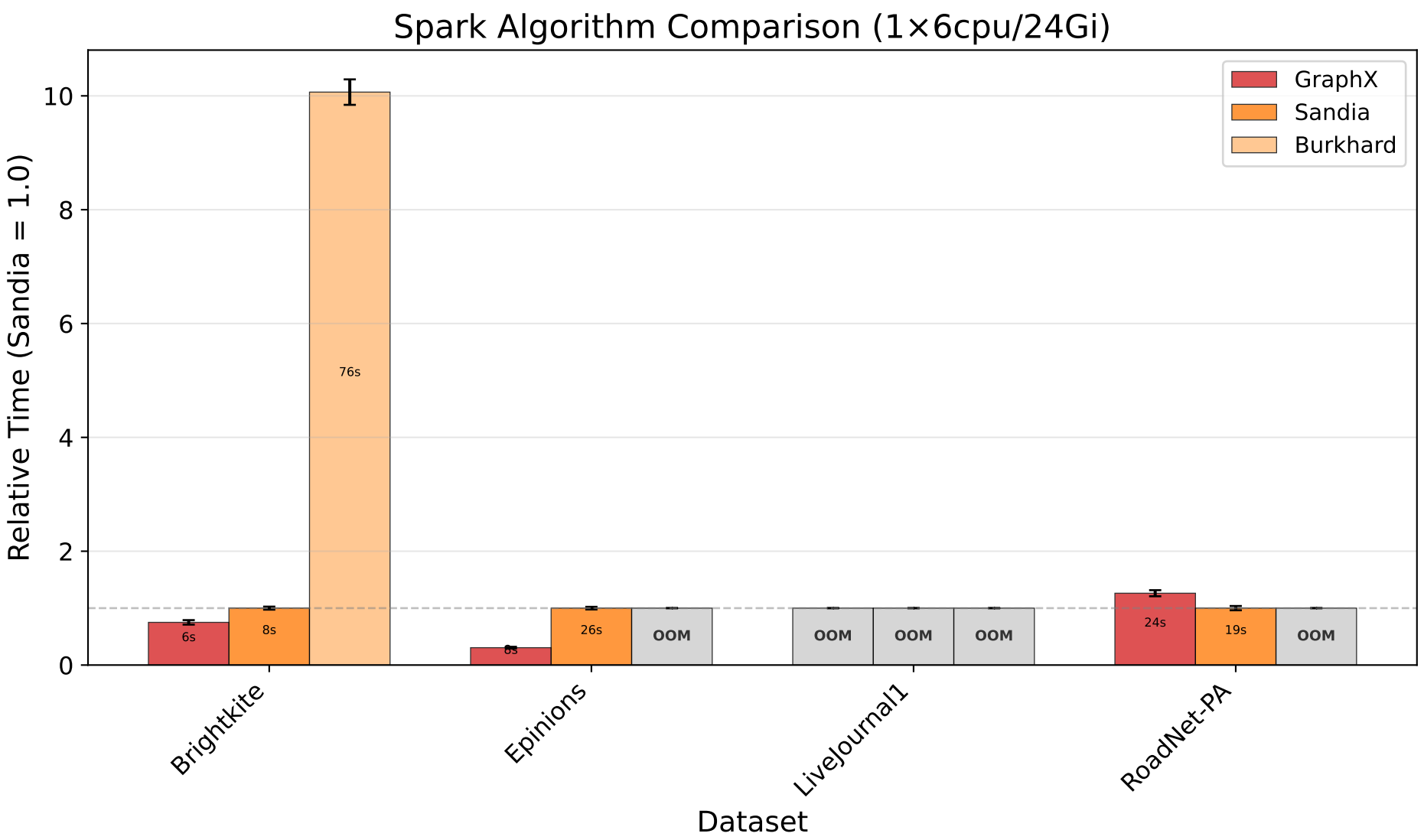
Датасет	Sandia			Burkhard		
	1×6/24	2×3/12	3×2/8	1×6/24	2×3/12	3×2/8
Brightkite	✓ 7.6	✓ 7.0	✓ 6.8	✓ 76.5	✓ 81.0	✓ 77.0
Epinions	✓ 26.2	✓ 21.8	✓ 20.9	✗ OOM	✗ OOM	✗ OOM
RoadNet-PA	✓ 18.7	✓ 17.3	✓ 17.9	✗ OOM	✗ OOM	✗ OOM
LiveJournal	✗ OOM	✗ OOM	✗ OOM	✗ OOM	✗ OOM	✗ OOM

Датасет	GraphX		
	1×6/24	2×3/12	3×2/8
Brightkite	✓ 5.7	✓ 5.7	✓ 5.2
Epinions	✓ 8.0	✓ 6.8	✓ 6.8
RoadNet-PA	✓ 23.6	✓ 21.7	✓ 22.1
LiveJournal	✗ OOM	✗ OOM	✗ OOM

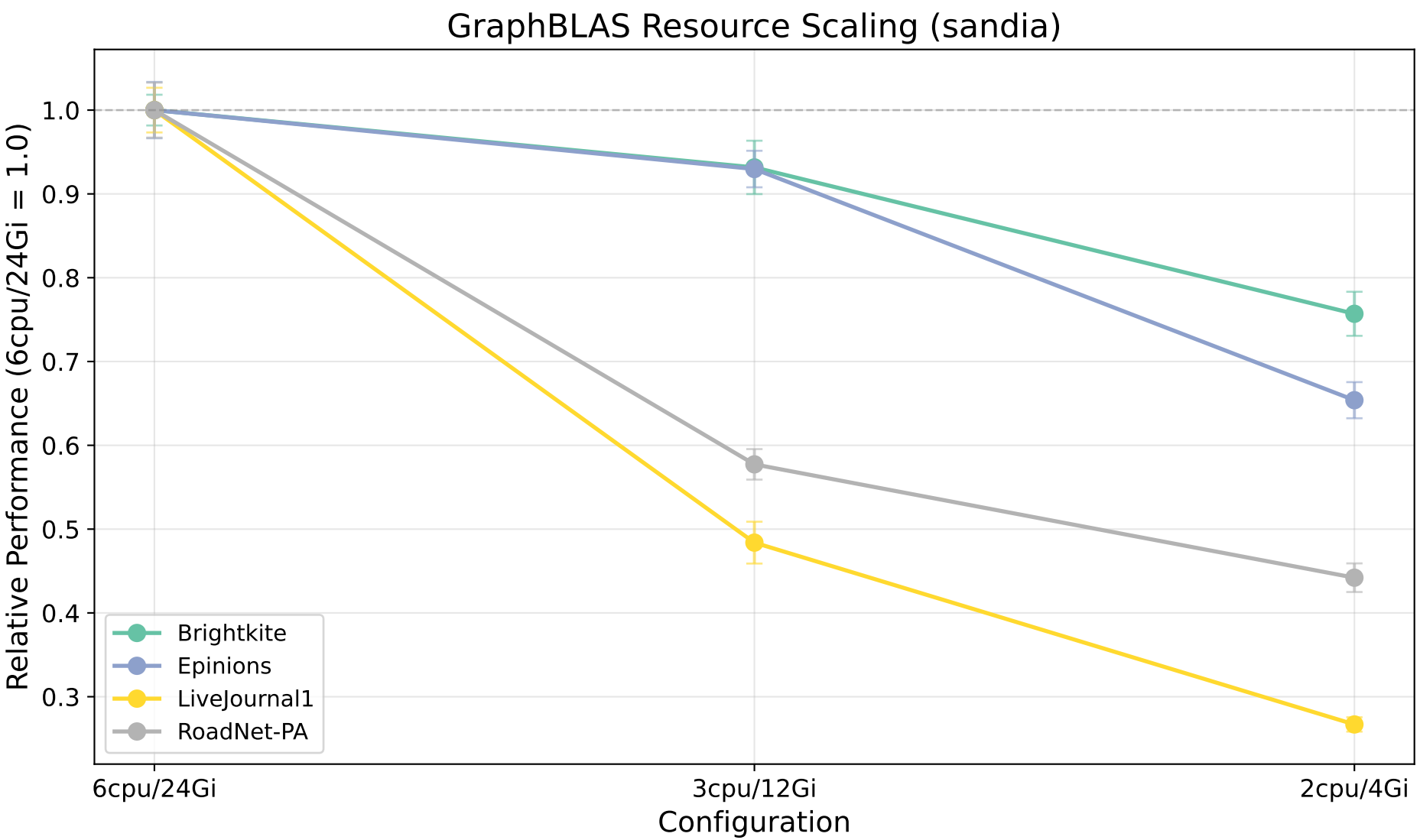
Сравнение алгоритмов GraphBLAS



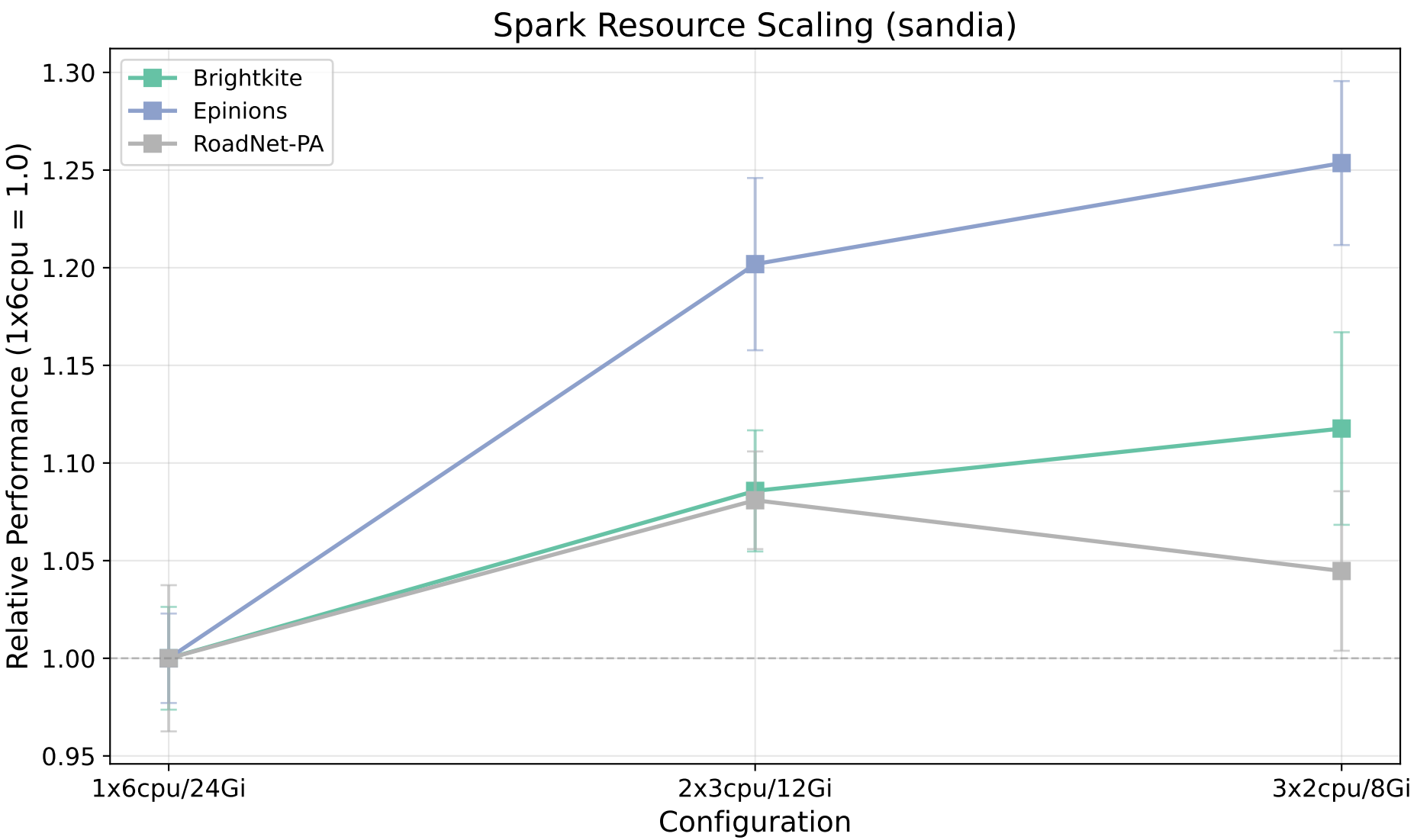
Сравнение алгоритмов Spark



Масштабирование ресурсов GraphBLAS



Масштабирование ресурсов Spark



Выводы

Результаты:

- Sandia быстрее Burkhard от 1.2 до 11 раз
- GraphX быстрее от 1.1 до 3.2 раз (кроме roadNet-PA: -20%)
- GraphBLAS быстрее Spark в 50-100× *на одной машине*
- GraphBLAS: OOM только Burkhard/2cpu на LiveJournal
- Spark: OOM на LiveJournal для всех конфигураций

Ключевые находки:

- Оверхед распределённой архитектуры Spark составляет 50-100× в условиях одного узла
- Для графов до 70M рёбер GraphBLAS является оптимальным выбором на single-node системах
- Spark Burkhard демонстрирует join explosion, требует оптимизаций
- **Вывод:** Выбор системы зависит от размера данных и доступной инфраструктуры

Почему Sandia оказалась быстрее?

Цель: Понять, какие свойства графов определяют производительность алгоритмов

Подход:

1. Анализ 16 графов: извлечение структурных свойств
2. Корреляционный анализ: связь свойств с ускорением (Sandia/GraphX)
3. Категоризация по степени ускорения
4. Построение предсказательной модели

Конфигурация:

- Spark 1×6cpu/24Gi
- 16 графов: от 6k до 2M вершин
- Алгоритмы: GraphX (встроенный) vs Sandia (самописный)

Обзор реализации: GraphX

Алгоритм: Пересечение множеств соседей для каждого ребра

```
// 1. Собрать соседей для каждой вершины
val nbrSets = graph.collectNeighborIds(EdgeDirection.Either)

// 2. Для каждого ребра (u,v): пересечь N(u) и N(v)
def edgeFunc(ctx) = {
  val (smallSet, largeSet) = if (ctx.src.size < ctx.dst.size)
    (ctx.src, ctx.dst) else (ctx.dst, ctx.src)
  val intersection = smallSet intersect largeSet
  ctx.send(intersection.size / 2)
}
```

Особенности реализации:

+ Оптимизация: пересекаем меньшее с большим множеством
+ Встроенная структура VertexSet (хеш-таблица)
– Требует сбор всех соседей заранее (memory intensive)
– Создаёт множество для каждой вершины, даже если пересечений нет

Обзор реализации: Sandia

Алгоритм: Матричное умножение $L \times L$ с маской L

```
// 1. Направить рёбра: оставить только i < j
val L = edges.filter({ (i,j) => i < j })

// 2. Соединение для поиска путей длины 2: (i,k,j)
val paths = L.map({(i,k) => (k, i)})
             .join(L.map({(k,j) => (k, j)}))

// 3. Соединение для проверки замыкания: (i,j) in L
val triangles = paths.map({(k, (i,j)) => (i, j)})
                     .join(L).count()
```

Особенности реализации:

+ Использует стандартные операции соединения (join)
+ Направление рёбер по ID (i < j) делит работу пополам
+ Разреженные структуры фильтруются естественным образом
– Создаёт промежуточный результат размером $\sum_k \binom{\deg(k)}{2}$
– Соединения зависят от распределения ключей (может быть неравномерным)

Результаты: категоризация по производительности

Таблица: **Sandia выигрывает** (Ускорение > 1.15)

Граф	**GraphX, сек**	**Sandia, сек**	**Ускорение**
RoadNet-TX	20.7	15.4	1.34×
RoadNet-PA	17.0	13.2	1.29×
CA-HepTh	2.26	1.79	1.26×
RoadNet-CA	25.1	20.0	1.26×
CA-GrQc	1.96	1.59	1.23×
Gnutella04	2.40	1.96	1.22×
Gnutella30	3.30	2.70	1.22×
AS-20000102	1.81	1.49	1.21×
Gnutella09	2.18	1.87	1.17×

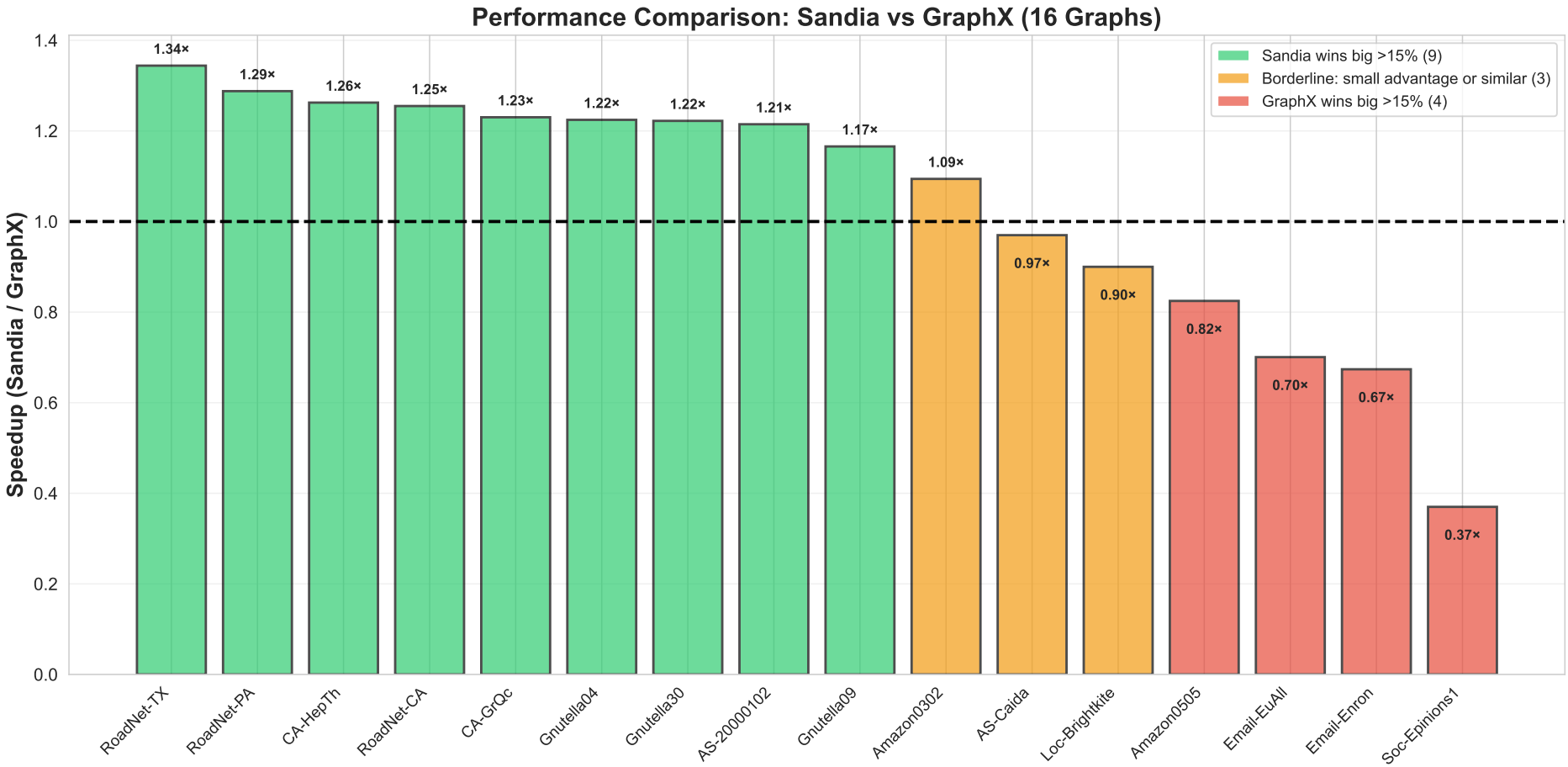
Таблица: **GraphX выигрывает** (Ускорение < 0.85)

Граф	**Speedup**
Soc-Epinions1	0.37×
Email-Enron	0.67×
Email-EuAll	0.70×
Amazon0505	0.82×

Таблица: **Borderline** (0.85–1.15)

Граф	**Ускорение**
Amazon0302	1.09×
AS-Caida	0.97×
Loc-Brightkite	0.90×

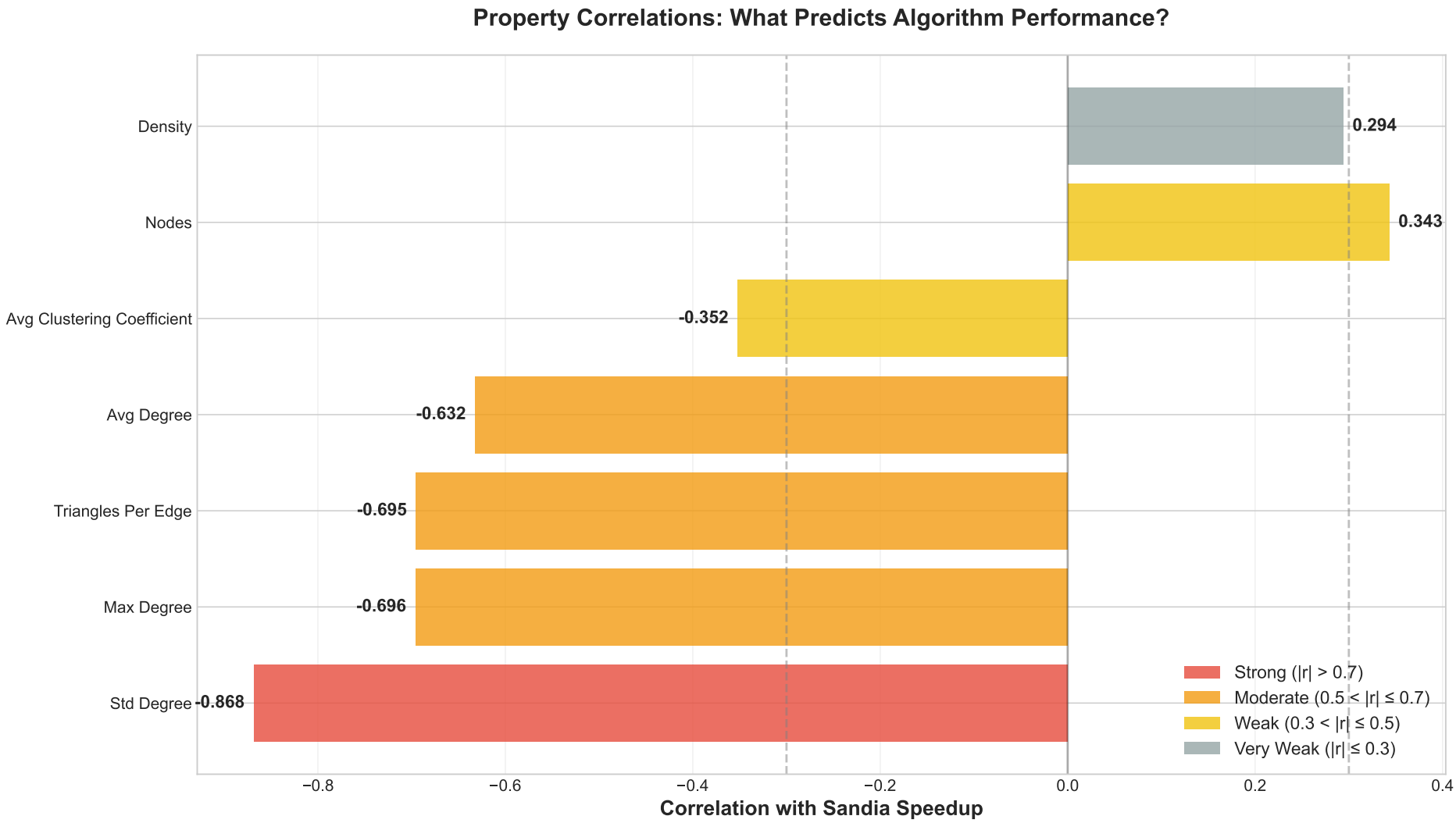
Performance: Sandia vs GraphX



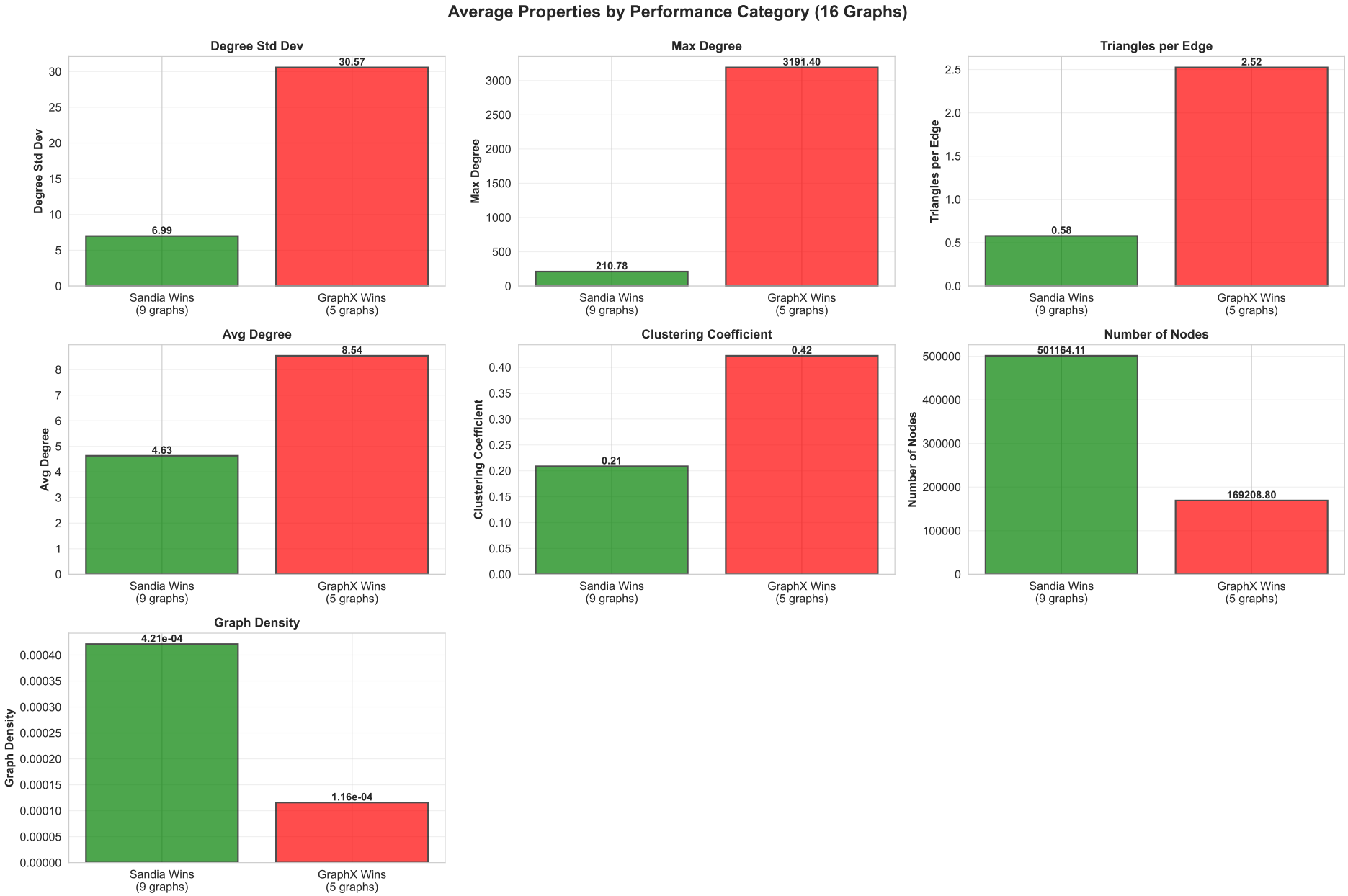
Ключевые наблюдения:

- Максимальное преимущество Sandia: **1.34×** (RoadNet-TX)
- Максимальное преимущество GraphX: **2.70×** (Soc-Epinions1)

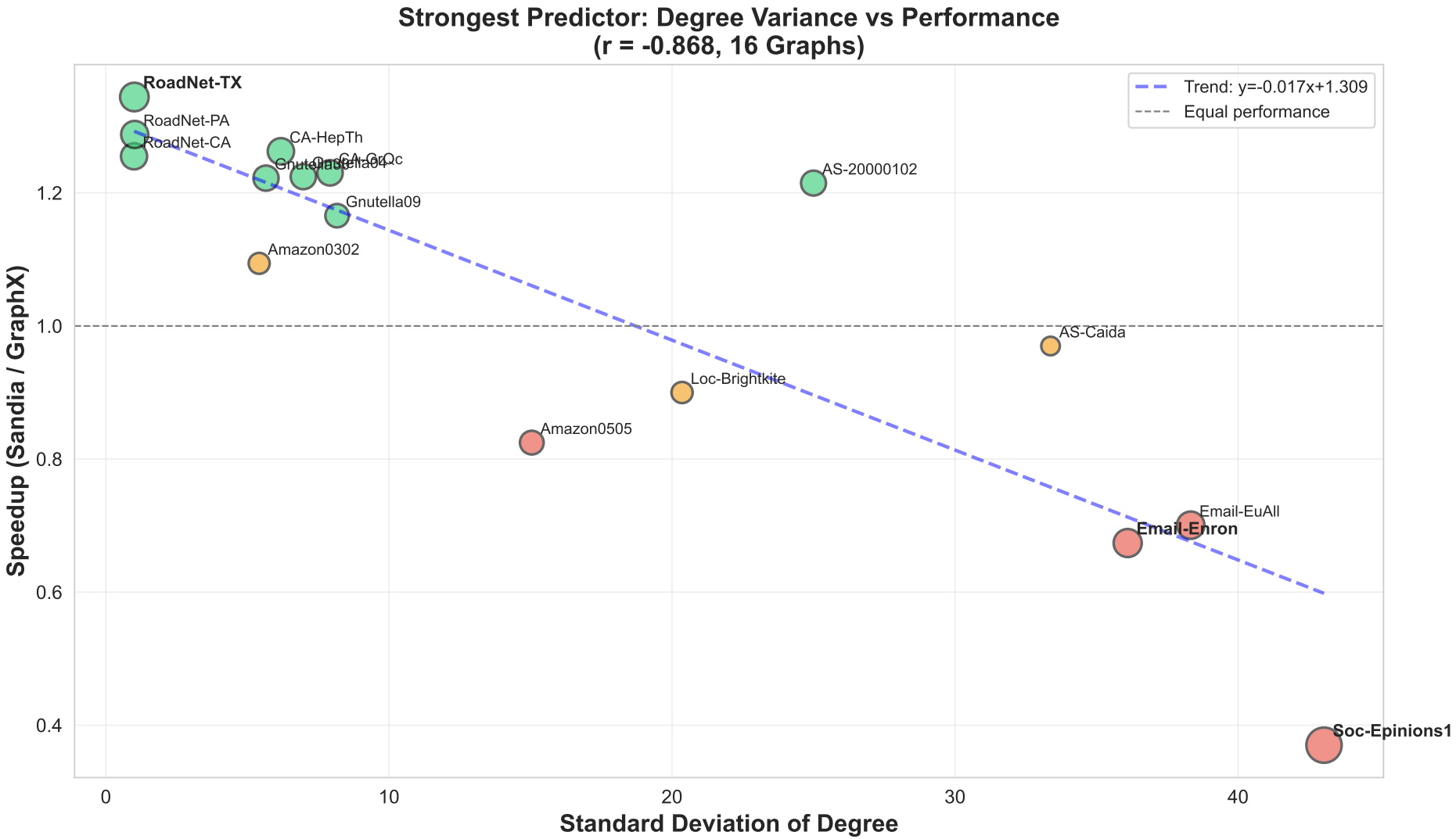
Корреляция свойств графа и ускорения Sandia



Средние значения по каждому свойству



Дисперсия степеней vs ускорение



Вывод: Sandia vs GraphX

Ключевые свойства графов:

1. **σ степеней** (ρ = -0.868): GraphX выигрывает при высокой дисперсии (> 20)
2. **Макс. степень** (ρ = -0.696): GraphX эффективен при наличии хабов (> 1000)
3. **Плотность треугольников** (ρ = -0.695): GraphX лучше на графах с высокой локальной связностью
4. **Средняя степень** (ρ = -0.632): Sandia оптимален при avg_degree < 10

Слабая корреляция: коэф. кластеризации (ρ = -0.352), число вершин ($\rho = +0.34$) и плотность ($\rho = +0.294$)

Вывод: Выбор алгоритма определяется в первую очередь **неравномерностью распределения степеней**, а не размером или плотностью графа.