УДК 004.62:004.67

СТРАТЕГИИ АДАПТИВНОГО РАСПАРАЛЛЕЛИВАНИЯ В АЛГОРИТМЕ СОРТИРОВКЕ ШЕЛЛА ДЛЯ ЗАДАЧ ПРЕДОБРАБОТКИ ДАННЫХ В BIG DATA СИСТЕМАХ

Л.В. Примакович
Ассистент кафедры экономической информатики БГУИР, магистрант
lyudmilaprimakoviych@yandex.by

И.П. Логинова
Старший научный сотрудник ОИПИ НАНБ, кандидат технических наук, доцент
irilog@mail.ru

Л.В. Примакович

Окончила Белорусский государственный университет информатики и радиоэлектроники. Область научных интересов связана с разработкой стратегий параллельных вычислений и применением методов высокопроизводительных вычислений для повышения эффективности обработки Big Data.

И.П.Логинова

Окончила Белорусский государственный университет, кандидат технических наук, доцент. Круг научных интересов: программирование, логическое проектирование и верификация цифровых схем, виртуализация, реализация параллельных алгоритмов.

Аннотация. В статье детально исследуются вопросы оптимизации параллельной реализации алгоритма сортировки Шелла, рассматриваемого как важный компонент этапа предобработки данных (ETL – Extract, Transform, Load) в конвейерах Big Data. Основное внимание уделено разработке и сравнению адаптивных стратегий распараллеливания, динамически меняющих степень распараллеливания в зависимости от текущего шага сортировки и объема данных. Анализируется влияние объема данных на производительность, а также обсуждаются методы оптимизации доступа к памяти в вычислительных системах с общей памятью, характерных для вычислительных узлов современных Big Data платформ. Предложена классификация стратегий и сформулированы практические рекомендации по повышению эффективности параллельных сортировок.

Ключевые слова: адаптивное распараллеливание, сортировка Шелла, балансировка нагрузки, оптимизация памяти, последовательность Ciura, многоядерные процессоры, производительность, предобработка данных, Big Data, ETL-процессы.

Введение. В условиях стремительно развивающихся многоядерных процессорных архитектур актуальной задачей становится организация эффективного распараллеливания традиционных последовательных алгоритмов.

В современных Big Data системах значительная часть времени выполнения аналитических запросов приходится на этапы предобработки и упорядочивания данных (ETL). Сортировка является одной из фундаментальных операций на этом этапе, используемой для подготовки данных к последующей агрегации, соединению или обучению моделей машинного обучения [1].

Неэффективная реализация сортировки может стать узким местом, сводя на нет преимущества распределенных вычислений даже при их выполнении на мощных кластерах.

Методика проведения сортировки. Сортировка Шелла является усовершенствованием простой сортировки вставками. Главная идея заключается в том, чтобы производить обмен элементов, находящихся далеко друг от друга, что значительно ускоряет процесс на

начальных этапах. В отличие от стандартной сортировки вставками, которая сравнивает соседние элементы, сортировка Шелла сравнивает элементы, отстоящие друг от друга на определенном расстоянии – шаге h. Процесс состоит из нескольких проходов. На каждом проходе с фиксированным шагом h массив разбивается на несколько независимых подмассивов (подпоследовательностей), каждый из которых сортируется методом вставок [2, с. 81-82]. Затем шаг уменьшается, и процесс повторяется. Финальный проход выполняется с шагом h=1, что эквивалентно обычной сортировке вставками, но к этому моменту массив уже почти упорядочен, поэтому финальная сортировка выполняется очень быстро.

Рассмотрим принцип работы на небольшом массиве.

На рисунке 1 показан процесс сортировки Шелла с начальным шагом равным 4. Видно, что для исходного массива сначала выполняется сортировка элементов, отстоящих друг от друга на 4 позиции, в ходе чего сортируется 4 подмассива с двумя числами в каждом. На следующем этапе шаг сортировки берется h=2, и сортируется 2 подмассива с четырьмя элементами в каждом. После этих двух этапов массив уже становится более упорядоченным, поэтому финальный проход с шагом h=1, т.е. обычная сортировка вставками, быстро завершает процесс.

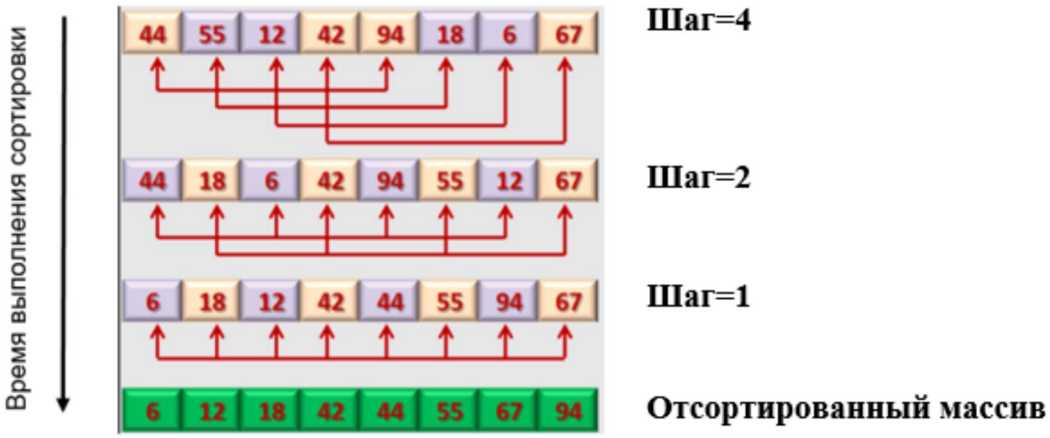

Рисунок 1. Визуализация процесса сортировки Шелла

Математически алгоритм можно описать следующим образом: для заданной последовательности шагов $h_1, h_2, \ldots, h_k$, где $h_k = 1$, на каждом проходе t выполняется сортировка вставками всех подмассивов вида $a[i], a[i+h_t], a[i+2h_t], \ldots$ для всех i от 0 до h_{t-1}.Эффективность и сложность алгоритма сортировки Шелла существенно зависит от выбора последовательности шагов. Классическая последовательность, предложенная самим Шеллом, имеет вид: $h_1 = n/2, \ldots, h_{i+1} = h_i/2, \ldots, h_k = 1$. Однако эмпирические исследования показали, что существуют более эффективные последовательности [3]. В частности, Марчин Цюра (Marcin Ciura) в 2001 году экспериментальным путем определил одну из наиболее эффективных известных последовательностей шагов для сортировки Шелла: [701, 301, 132, 57, 23, 10, 4, 1].

Данная последовательность была получена в результате масштабных эмпирических исследований и демонстрирует лучшую производительность для широкого диапазона размеров массивов по сравнению с классической последовательностью Шелла. Теоретическое обоснование эффективности последовательности Ciura связано с оптимизацией двух противоречащих друг другу факторов: большие шаги быстрее перемещают элементы на нужные позиции, но требуют больше сравнений; маленькие шаги выполняют меньше сравнений, но медленнее упорядочивают массив. Последовательность Ciura находит оптимальный баланс между этими факторами, минимизируя общее количество операций сравнения и перемещения элементов. Для массивов размером более 701 элемента

последовательность продолжается эмпирическим правилом $h_{k-1}=|h_k*2.2|$. Фиксированный и относительно небольшой набор шагов (8 для большинства практических задач) делает её идеальной для параллельных реализаций, так как позволяет точно предсказать количество итераций и объем работы на каждой из них.

Стратегии распараллеливания в сортировке Шелла. Структура алгоритма Шелла, основанная на независимой сортировке подмассивов на каждом шаге, создает естественные возможности для параллельных вычислений. Однако так называемый «наивный» подход к распараллеливанию вычислений, при котором все подмассивы на всех шагах сортируются параллельно, неэффективен. При малых значениях h (например, 4, 2, 1) количество подмассивов невелико, а объем работы в каждом из них мал. Затраты на создание и синхронизацию потоков в таком случае превышают выигрыш от параллельного выполнения.

В рамках исследования были реализованы и протестированы три стратегии, отличающиеся степенью адаптации к текущему шагу.

1. «Наивное» распараллеливание.

Все подмассивы на всех шагах сортировки распределяются между всеми доступными потоками. Эта стратегия служит точкой отсчета для демонстрации проблемы «наивного» подхода. На рисунке 2 наглядно показано, как распределялись бы задачи между потоками на каждом шаге при условии доступности четырех потоков в системе:

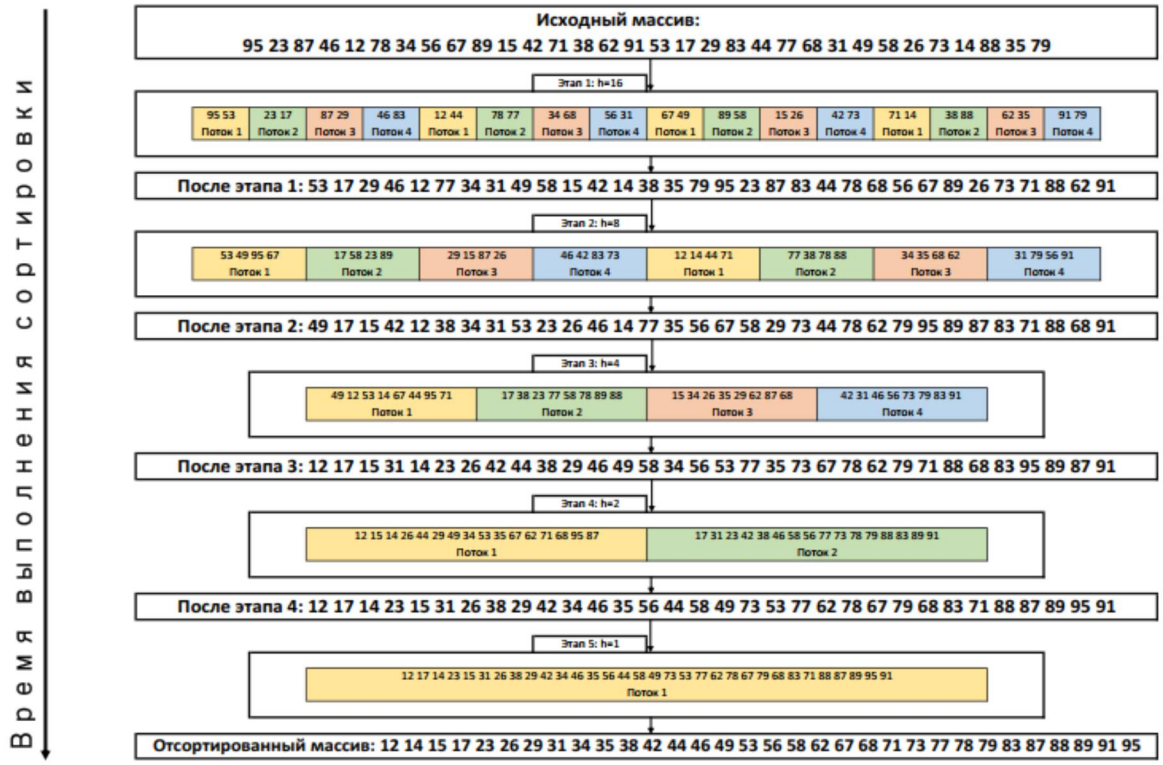

Рисунок 2. Схематическое представление «наивного» распараллеливания

2. Пороговое распараллеливание (распараллеливание с большими шагами).

Данная стратегия активирует параллельную обработку только тогда, когда выполняется условие $h \geq num\_threads*4$, где h – текущий шаг сортировки, а num_threads – количество доступных потоков.

Хотя эта стратегия проста в реализации, она может оказаться недостаточно гибкой, оставляя неиспользованными вычислительные ресурсы на средних шагах сортировки, когда h находится в диапазоне между num_threads и num_threads*4.

На рисунке 3 наглядно показано, как распределялись бы задачи между потоками на каждом шаге при условии доступности четырех потоков в системе:

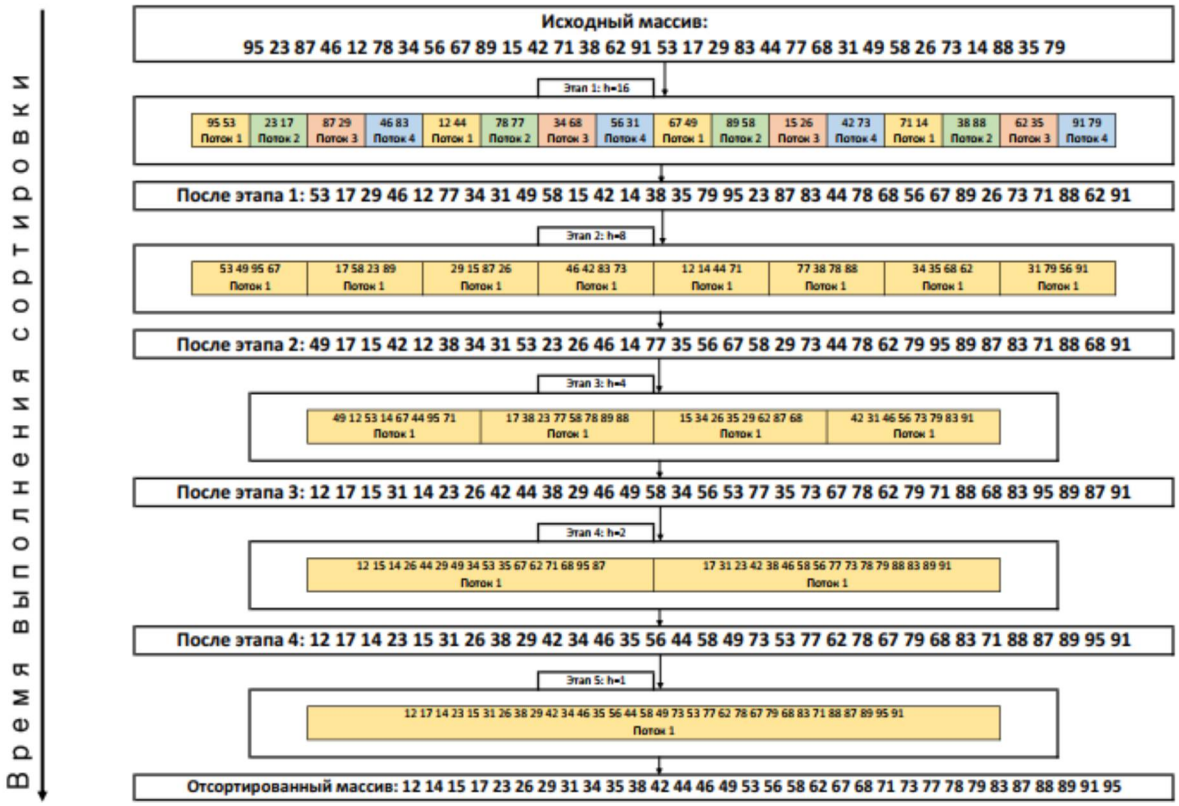

Рисунок 3. Схематическое представление порогового распараллеливания

3. Динамическое управление числом потоков (ручная балансировка).

В этой стратегии количество задействованных потоков напрямую привязывается к размеру текущего шага h. Алгоритм реализует три режима работы:

- при $h \geq max\_threads*2$ используются все доступные потоки (max_threads);
- при $max\_threads \leq h < max\_threads*2$ создается ровно h задач (по числу подмассивов), которые распределяются между свободными и освободившимися в процессе работы потоками;
- при $h < max\_threads$ выполняется последовательная обработка.

Данная стратегия показала наилучшие результаты в экспериментах, максимально эффективно используя вычислительные ресурсы на всех этапах сортировки. Ее графическое представление показано на рисунке 4 при тех же условиях, что и для рассмотренных ранее стратегий.

Цель данной работы – количественно оценить эффективность различных стратегий адаптивной организации параллельных вычислений на базе алгоритма сортировки Шелла и подтвердить теоретические выводы о превосходстве динамического подхода. В ходе работы были также изучены возможности двухуровневого распараллеливания. Эта достаточно сложная стратегия предполагает проведение распараллеливания на двух уровнях: на верхнем уровне различные шаги из последовательности обрабатываются параллельно разными потоками, а внутри обработки каждого достаточно большого шага ($h \geq 4$) параллельно обрабатываются отдельные подмассивы. Стратегия особенно эффективна при использовании последовательности Ciura, которая содержит фиксированное небольшое количество шагов. Однако ее реализация возможна на специализированных архитектурах, но в рамках данной

работы было решено сосредоточиться на стратегиях, дающих оптимальное соотношение эффективности и сложности реализации.

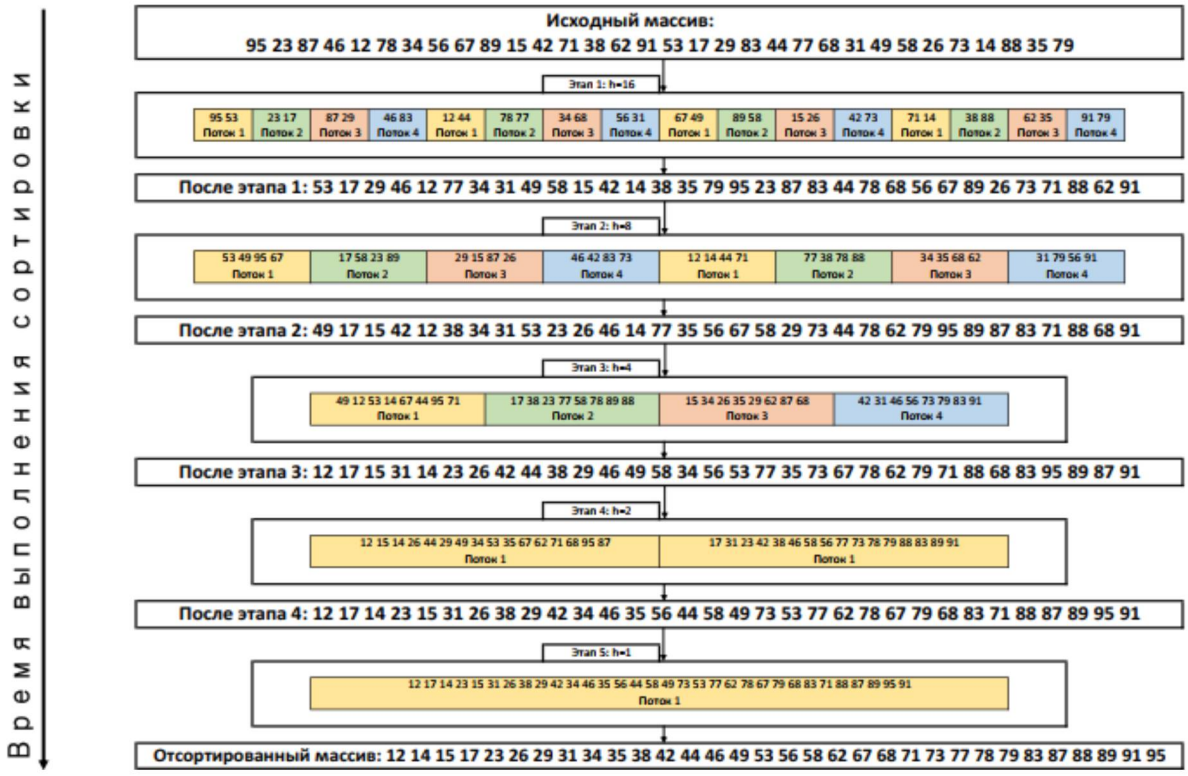

Рисунок 4. Схематическое представление адаптивного распараллеливания

Критически важным моментом в любой параллельной реализации является барьерная синхронизация после завершения каждого шага сортировки. Алгоритм Шелла требует, чтобы все подмассивы для текущего шага h были полностью отсортированы до того, как алгоритм перейдет к следующему шагу. Это гарантирует корректность состояния данных для следующей итерации [4]. В реализациях с использованием OpenMP и PPL синхронизация обеспечивается автоматически в конце параллельных регионов за счет использования конструкции #pragma omp parallel или вызова parallel_for с неявной синхронизацией по завершении. При использовании низкоуровневых средств (Windows API) синхронизация реализуется вручную с помощью примитивов, таких как барьеры (Barrier) или ожидание завершения всех потоков (WaitForMultipleObjects).

Проведение экспериментальных расчетов. Реализация и тестирование программы, использующей каждый из рассмотренных вариантов распараллеливания, производились на ноутбуке с характеристиками:

1. Процессор: AMD Ryzen 5 4600H с графикой Radeon
    a) Текущая скорость: 1.38 ГГц (в 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

Для проведения тестирования была разработана серия реализаций алгоритма по каждой из трех стратегий распараллеливания на языке C++. Компиляция осуществлялась средствами MSVC (Microsoft Visual C++) с активированной поддержкой спецификации OpenMP 2.0., что

позволило задействовать директивный подход к распараллеливанию там, где это было уместно. Во всех параллельных реализациях максимальное количество используемых потоков устанавливалось равным 12, что соответствует числу логических процессоров целевой системы. Программно это значение определялось через вызов стандартной функции omp_get_max_threads() для OpenMP-реализаций и через GetMaximumProcessorCount() при использовании Windows API, что обеспе-чивало адаптацию алгоритма к аппаратным возможностям тестового стенда (т.е. другой аппаратной платформы) без изменения исходного кода. Чтобы исключить влияние сложности типов данных на результаты, все замеры производились на массивах целых чисел (int). Для тестирования использовались простые целочисленные массивы различных размеров (от 10000 до 200000 элементов). Каждый тест включал проверку корректности сортировки и измерение времени выполнения с высокой точностью. Корректность работы каждой реализации проверялась после каждого эксперимента путем полного перебора отсортированного массива и сравнения каждой пары соседних элементов (условие array[i] <= array[i+1] для всех i). В случае нарушения порядка тест считался проваленным, и его результаты исключались из выборки. Это гарантирует, что все приведенные в таблице 1 данные относятся к алгоритмам, успешно выполнившим сортировку. Начальный набор сгенерированного неотсортированного массива для каждой стратегии был одинаков в целях избегания лучших результатов за счет большей упорядоченности начальных значений. В качестве основных метрик производительности измерялись время выполнения, коэффици-ент ускорения $Sp=T_1/T_p$ (где T_1 – время последовательного выполнения, T_p – время параллельного выполнения на p потоках) и эффективность использования потоков: $E_p=Sp/p$. Реализованные стратегии были протестированы на различных объемах данных. Эффективность адаптивных стратегий, рассчитанная на 12 потоках, в сравнении с базовым подходом (котором все подмассивы на всех шагах сортируются параллельно) иллюстрирует Таблица 1 на примере массива в 100 000 элементов (реализация на OpenMP, целочисленные данные).

Таблица 1. Влияние стратегии на эффективность алгоритма

Стратегия	**Описание распараллеливания**	**Sp_{max}**	**E_p, %**	**Ключевой критерий переключения**
Базовая (все подмассивы)	Параллельная обработка всех подмассивов на каждом шаге h	2.22	17.0	–
Пороговая	Параллельные вычисления только для больших h	2.22	17.0	h >= num_threads*4
Адаптивная (динамическая)	Изменение числа задействованных потоков в зависимости от h	3.70	30.8	При h >= max_threads*2 используются все потоки; при h >= max_threads используется h потоков; иначе – последовательная

Как видно из таблицы 1, стратегия динамического управления потоками продемонстрировала наилучшие результаты, обеспечив ускорение в 3.70 раза по сравнению с последовательной реализацией. Дальнейшие эксперименты с использованием других технологий параллельного программирования подтвердили общие закономерности. Так, реализация адаптивной стратегии с использованием Parallel Patterns Library (PPL) в C++ дала схожий результат – ускорение достигло значения 3.92 на 200 тысячах элементов при использовании 12 потоков. Реализация на Windows API, несмотря на сложность ручного управления потоками и синхронизацией, также подтвердила выявленные закономерности: пиковая производительность достигается при количестве потоков, равном или меньшем количества физических ядер процессора (6 в данном случае).

Оптимизация доступа к данным. В контексте параллельной сортировки для Big Data особое значение приобретает оптимизация доступа к памяти. Многоядерные процессоры современных вычислительных систем обладают сложной иерархией памяти, включающей

несколько уровней кэша. Неэффективный доступ к памяти может стать узким местом, стирая преимущества параллельного выполнения. При обработке больших массивов данных, значительно превышающих объемы кэш-памяти, проблема оптимизации доступа к памяти становится наиболее значимой. Дополнительно проведенные эксперименты с разными типами данных четко показали, что сортировка по простым числовым полям выполняется в 1.5-2 раза быстрее, чем по строковым. Это объясняется более простой и предсказуемой для процессора операцией сравнения чисел, лучшей локализацией данных и работой кэш-памяти. Для систем Big Data это означает, что использование идентификационных числовых ключей вместо строковых данных при сортировке может дать существенный выигрыш в производительности на этапе предобработки. В процессе проведения серии экспериментов было выявлено, что использование последовательности Ciura вместо классической (n/2, n/4...) также является оптимизацией, снижающей общее количество операций сравнения и перемещений в памяти [5]. В масштабах Big Data, где сортируются терабайты данных, даже незначительное снижение количества операций дает колоссальный суммарный эффект. Еще одним важным наблюдением стало то, что при использовании адаптивных стратегий эффективность использования потоков (отношение достигнутого ускорения к теоретически максимальному) остается на приемлемом уровне (25-35%) даже при максимальном количестве логических процессоров, в то время как для базовых стратегий этот показатель падает ниже 20%. При работе с Big Data в кластерных системах это важно виду того, что кластерные системы должны максимально задействовать ресурсы каждого узла, чтобы обеспечивать линейную масштабируемость при разрастании кластера.

Анализ реализаций показал, что следующие методы оптимизации доступа к памяти оказывают существенное влияние на производительность:

1. Локализация данных – обеспечение последовательного доступа к элементам массива в пределах одного подмассива улучшает использование кэш-памяти процессора. При работе с Big Data, когда массив не помещается в кэш, это позволяет минимизировать простои процессора и врямя доступа к данным.

2. Предварительная выборка данных (prefetching) – современные компиляторы и процессоры способны загружать необходимые данные в кэш заранее, что особенно эффективно при регулярных шаблонах доступа, характерных для сортировки Шелла. В Big Data системах это свойство может быть использовано для упреждающей подгрузки данных из медленной памяти в быструю.

3. Минимизация ложного разделения кэш-линий (false sharing) – при параллельной обработке соседних подмассивов разными потоками может возникать конкуренция за общие кэш-линии, что снижает производительность. Эффективные стратегии распараллеливания минимизируют проблему ложного разделения кэш-линий путем правильного распределения работы между потоками, что особенно актуально для многопоточных серверов, обрабатывающих Big Data.

На основе проведенного исследования можно сформулировать следующие практические рекомендации для разработчиков параллельных реализаций алгоритма сортировки Шелла:

1. Использование адаптивных стратегий: предпочтение следует отдавать стратегиям динамического управления количеством потоков в зависимости от текущего шага сортировки.

2. Применение последовательности Ciura, т.к. использование эмпирически оптимальной последовательности шагов обеспечивает лучшую производительность по сравнению с классической последовательностью Шелла.

3. Оптимизация доступа к памяти. Следует уделять внимание локализации данных и минимизации конфликтов при доступе к памяти из разных потоков.

4. Выбор технологии для организации параллельных вычислений. Для большинства задач оптимальным выбором являются высокоуровневые библиотеки параллельного программирования (OpenMP, PPL, TPL), которые обеспечивают хороший баланс между производительностью и сложностью разработки [6].

Заключение. Проведенный анализ доказал, что ключом к высокой эффективности параллельной сортировки Шелла является не столько выбор конкретной библиотеки многопоточности, сколько использование интеллектуальных адаптивных стратегий, тонко реагирующих на параметры алгоритма на каждом этапе его работы. Наиболее эффективной оказалась стратегия динамического управления числом потоков в зависимости от текущего шага сортировки, которая позволила достичь ускорения до 3.92 раза по сравнению с последовательной реализацией. Использование оптимизированных последовательностей шагов, таких как последовательность Ciura, показало себя лучшей практикой при реализации алгоритма сортировки Шелла. Результаты проведенного исследования носят общий характер и могут быть применены для оптимизации других параллельных алгоритмов, обладающих схожей структурой вложенных независимых задач, а также для настройки параметров выполнения в распределенных вычислительных средах. Дальнейшие исследования могут быть направлены на изучение гибридных стратегий распараллеливания, сочетающих преимущества различных подходов, а также на оптимизацию работы с памятью для специализированных аппаратных архитектур, используемых в дата-центрах.

Список литературы

[1] Седжвик, Р. Алгоритмы на C++. Фундаментальные алгоритмы и структуры данных : в 2 ч. / Р. Седжвик ; пер. с англ. – Москва : ДиаСофтЮП, 2002. – Ч. 1–2. – 688 с.

[2] Вирт, Н. Алгоритмы и структуры данных. Новая версия для Оберона + CD / Н. Вирт ; пер. с англ. Ф. В. Ткачев. – М. : ДМК Пресс, 2010. – 272 с. – ISBN 978-5-94074-584-2.

[3] Ciura, M. Best Increments for the Average Case of Shellsort / M. Ciura // Proceedings of the 13th International Symposium on Fundamentals of Computation Theory. – 2001. – P. 106–117.

[4] Гергель, В. П. Теория и практика параллельных вычислений / В. П. Гергель. – Москва : Бином. Лаборатория знаний, 2007. – 424 с.

[5] Таненбаум, Э. Архитектура компьютера / Э. Таненбаум, Т. Остин. – 6-е изд. – Санкт-Петербург : Питер, 2013. – 816 с.

[6] Кэтлин, Дж. Конкурентность в C#. Асинхронное, параллельное и многопоточное программирование / Дж. Кэтлин. – Санкт-Петербург : Питер, 2020. – 288 с.

Авторский вклад

Примакович Людмила Васильевна – программная реализация алгоритмов на C++, проведение экспериментов, сбор и обработка результатов, анализ влияния факторов на производительность.

Логинова Ирина Петровна – постановка задачи исследования, формулировка идеи адаптивного распараллеливания для Big Data, верификация полученных выводов.

ADAPTIVE PARALLELIZATION STRATEGIES AND MEMORY ACCESS OPTIMIZATION IN THE SHELLSORT ALGORITHM FOR DATA PREPROCESSING TASKS IN BIG DATA SYSTEMS

L.V. Primakovitch
Assistant at the Department of Economic Informatics, BSUIR, Master's Degree Student
lyudmilaprimakoviych@yandex.by

I.P. Loginova
Senior Researcher at JIIT NASB, Candidate of Technical Sciences, Associate Professor
irilog@mail.ru

Abstract.. The article thoroughly investigates the optimization of parallel implementation of the Shell sort algorithm, considered as an important component of the data preprocessing stage (ETL) in Big Data pipelines. The focus is on the development and comparison of adaptive parallelization strategies that dynamically change the degree of parallelism depending on the current sorting step and data volume. The influence of data volume on performance is analyzed, and memory access optimization methods in shared memory computing systems, typical for computing nodes of modern Big Data platforms, are discussed. A classification of strategies is proposed and practical recommendations for improving the efficiency of parallel sorting are formulated.

Keywords:. adaptive parallelization, ll sort, load balancing, memory optimization, Ciura sequence, multi-core processors, performance, data preprocessing, Big Data, ETL processes.