Санкт-Петербургский государственный университет
Кафедра компьютерного моделирования и
многопроцессорных систем

Свешникова Светлана

Магистерская диссертация

Методы повышения производительности Apache Spark на системах с неоднородной памятью

Направление 010300
«Фундаментальная информатика и информационные технологии»
Магистерская программа «Вычислительные технологии»

Научный руководитель
Ганкевич И.Г.

Санкт-Петербург
2018

Содержание

1 Введение 4
1.1 Apache Spark 4
1.2 Разные виды памяти 5
1.3 Использование Big Data фреймворков на HPC системах . . . 5
1.3.1 Оптимизация Big Data фреймворков под HPC системы 6
1.3.2 Утилиты для запуска Big Data приложений на HPC системах 8
1.3.3 Системы HPC + Big Data от вендоров 9
1.3.4 Примеры решения задач в связке Big Data + HPC . . 9
1.4 Способы работы с NUMA 9
1.4.1 Привязка потоков к процессорам 10
1.4.2 Размещение данных в памяти 11
2 Обзор литературы 12
3 Постановка задачи 15
4 Архитектура памяти Apache Spark 17
5 Исследование производительности на Intel Xeon Phi KNL 21
5.1 Выбор методов и тестирование 21
5.2 Анализ результатов 25
6 Исследование производительности на узле wombat-3 26
6.1 Выбор методов и тестирование 26
6.2 Анализ результатов 28

7 Выводы 30

8 Заключение 32

9 Глоссарий 33

Список иллюстраций 35

Список таблиц 36

Список литературы 37

1 Введение

1.1 Apache Spark

Apache Spark — фреймворк для обработки больших объемов данных в оперативной памяти. Позволяет писать эффективный код, выполняющий параллелизм по данным. Для Spark написаны различные библиотеки, позволяющие ему быть востребованным во многих областях: машинное обучение (библиотека Mlib), потоковая обработка данных (Apache Spark Streaming), выборка данных из таблиц (SparkSQL), обработка графов (GraphX). Apache Spark входит в Hadoop-экосистему — набор инструментов для выполнения полного цикла работы с большими данным: извлечение данных из исходного источника, очистка и преобразование данных, обработка, представление результата (визуализация, передача в другую систему, запись в базу данных и т.д.) При разработке начального продукта экосистемы — Apache Hadoop разработчики делали упор на возможность обработки большого количества данных на недорогих доступных средней фирме кластерах типа Beowulf. Особенности: стандартная архитектура, обычная не высокоскоростная сеть Ethernet, API для языков программирования, активно используемых в разработке коммерческих приложений: Java, Python. В процессе развития была создана целая экосистема, позволяющая удобно выполнять весь цикл обработки данных от получения до представления результатов.

1.2 Разные виды памяти

1.3 Использование Big Data фреймворков на HPC системах

HPC (High-Perfomance Computing) — приложения, вычислительные системы и их окружение, специализирующиеся на вычислениях требующих интенсивного использование процессора и памяти (высокая частота обращений, скорость ответа). Обычно это программы, выполняющие различные научные расчеты. При этом в HPC нет достаточно универсального сложившегося стека программ и технологий, позволяющего выполнять полный цикл обработки данных. В связи с этим возможность запуска на HPC кластерах программ экосистемы Hadoop представляется очень перспективным направлением развития. Вместе с тем есть проблемы внедрения и использования big-data приложений на таких системах.

Запуск Apache Spark на таком железе дает дополнительные возможности ускорения работы фреймворка, которые ранее не были востребованы. Одной из таких возможностей является использование неоднородной памяти. Неоднородная память — схема реализации памяти компьютера, когда время доступа к данным в памяти определяется ее расположением по отношению к процессору. Это архитектурное решение нечасто используется на обычных коммерческих кластерах, так как создает дополнительные сложности при написании кода, но при правильном использовании способно существенно увеличить скорость и эффективность выполнения программ. К неоднородной памяти также можно отнести и использование на кластерах

сопроцессоров и видеокарт.

В таблице 1 приведены основные отличия в решениях для кластеров ипользуемых в задачах Big Data и HPC [1]. Из таблицы видно, что есть отличия как в архитектурных решениях так и в способах работы программ и пользователей. В итоге возникают следующие вопросы:

- запуск задач через планировщик HPC кластера,
- оптимизация Big Data фреймворков под архитектурные и программные решения HPC кластера.

В ходе обзора литературы были найдены материалы, которые условно можно разделить на следующие темы:

- оптимизация Big Data фреймворков под HPC системы,
- утилиты для запуска Big Data приложений на HPC системах,
- системы HPC + Big Data от вендоров,
- примеры решения задач в связке Big Data + HPC.

1.3.1 Оптимизация Big Data фреймворков под HPC системы

На официальном сайте компании Intel есть целый раздел, посвященный разработкам компании в области Big Data и в частности — в сфере оптимизации приложений стека Apache Hadoop. Среди разработок — оптимизация Java и Hadoop под процессоры Intel Xeon [2], библиотека машинного обучения BigDL, призванная заменить MLib, вызывающая функции библиотеки Intel DAAL.

Компонент	Big Data	HPC
СХД	Кластер использует систему хранения данных с прямым подключением. Основная файловая система — HDFS.	Кластер использует в основном сетевую файловую систему.
Сеть	Ethernet	Высокоскоростные протоколы соединения — Infiniband, Omni-Path
Основные языки разработки запускаемых приложений	Java, Scala, Python	C/C++, Fortran, Python
Планировщик	Используется для управления работой задач в Hadoop-среде. Примеры — YARN, Mesos, FairScheduler	Используется для запуска любых задач пользователя. Примеры — PBS, Torque, Sun Grid Engine.
Вычисления	Основная часть вычислений производится на тех узлах, где лежат данные. Пересылка данных по сети сводится к минимуму.	Зависит от реализации конкретного приложения.

Таблица 1: Сравнение технологий, используемых в Big Data и HPC.

1.3.2 Утилиты для запуска Big Data приложений на HPC системах

Действительно мощные HPC кластера — суперкомпьютеры, не имеющие серийных решений. Это уникальные компьютеры с уникальной архитектурой, которую необходимо учитывать при создании утилит для запуска задач Hadoop-стека. Мной были рассмотрены две статьи с такими решениями. Статья [3] описывает утилиту MyHadoop, созданную в суперкомпьютерном центре Сан-Диего. Myhadoop — инструмент для запуска hadoop-job на HPC кластерах, через планировщики задач. Поддерживает SLURM, Torque, Sun Grid Engine. Умеет работать с интерфейсом IPoIB(IP over InfiniBand). Похожее решение для Apache Spark описано в статье [4] специалистами суперкомпьютерного центра Барселоны для компьютера MareNostrum. Предложенный фреймворк Spark4MN содержит в себе 3 функциональности:

- разворачивание кластеров Spark и запуск пользовательских задач,
- режим бенчмарка — пользовательская задача выполняется несколько раз на различных конфигурациях железа,
- утилита для рисования графиков, отражающих результаты запуска бенчмарка.

1.3.3 Системы HPC + Big Data от вендоров

Известный разработчик и поставщик суперкомпьютеров компания Cray разработала специальные пакеты программного обеспечения Urika-XC и Urika-GX, создающие среду для запуска Big Data приложений на своих системах. Поддерживаемые приложения: Apache Spark, Apache Hadoop, Intel BigDL, Cray Graph Engine.

1.3.4 Примеры решения задач в связке Big Data + HPC

Одной из задач, требующих не только большого количества вычислений но и обработки большого объема данных является задача сборки генома. Специалисты Национального Центра Биотехнологий США попробовали решить эту задачу с использованием специализированного пакета для вычислений в биоинформатике Myrna и Intel Apache Hadoop [5].

1.4 Способы работы с NUMA

Есть 2 основных подхода используемых для организации работы с NUMA памятью: привязка потоков к процессорам и правильное размещение данных в памяти. В данной работе применяется первый метод.

1.4.1 Привязка потоков к процессорам

Вычислительные потоки распределяются по ядрам с помощью специального системного планировщика. Планировщик решает на каком ядре в конкретный момент времени будет обрабатываться поток. В процессе работы планировщик может перемещать поток между ядрами. Предположим, что у нас есть 2 узла. Поток начал выполнение на ядре узла 1 и в его локальной памяти сохранил необходимые для работы данные. Затем поток прервал выполнение ядре узла 1 и планировщик перенес его выполнение на ядро узла 2. Теперь, когда потоку понадобятся используемые им данные он будет вынужден запрашивать их из удаленной памяти узла 2. Возросшее время доступа к памяти часто перекрывает преимущества от дополнительного времени выполнения, полученного путем использования ресурсов другого узла.

Привязка потоков гарантирует, что при выборе ядра для потока выполнения планировщик не будет рассматривать ядра другого узла. Тем самым гарантируется, что в процессе выполнения данные, необходимые потоку всегда будут находиться в локальной памяти.

К недостаткам этого метода можно отнести то, что он ограничивает работу планировщика. Планировщик не может отдать неиспользуемые ядра из одной группы ожидающим очереди на выполнение потокам из другой группы. Это провоцирует борьбу за ресурсы и может увеличить общее время выполнения приложения. В целом, эффективность метода зависит от вычислительной задачи.

1.4.2 Размещение данных в памяти

Существует 2 способа размещения данных в памяти. Используемый способ размещения зависит от операционной системы.

1. Локальность по отношению к первому доступу. Операционная система ожидает непосредственного обращения к памяти, чтобы выделить участок.

2. Локальность по отношению к запросу. Операционная система приписывает страницы виртуальной памяти тем участкам памяти, которые относятся к запрашиваемому узлу.

В случае, когда требуется предсказуемое распределение памяти в многопоточном приложении, работающем на нескольких узлах, хорошим решением послужит использование библиотеки libnuma. Используя API этой библиотеки программист может закрепить диапазон адресов виртуальной памяти за определенным узлом. Недостатком является дополнительная работа для программиста по планированию распределения памяти и размещения данных.

2 Обзор литературы

Apache Spark является популярным инструментом во многих областях. Он позволяет выполнять вычисления, работать с графами и таблицами, может применяться в области машинного обучения. Производительность фреймворка измерялась разными исследователями на разных примерах. Здесь приведены некоторые из них. При рассмотрении работ внимание уделяется в первую очередь аспекту использования NUMA.

В работе [6] группа специалистов из IBM тестирует производительность одного из компонентов Apache Spak — библиотеки GraphX, используемой для эффективных вычислений на графах. Запуск осуществляется на кластере содержащем 17 узлов с процессорами IBM POWER8. Используются стандартные версии Java и Apache Spark, не содержащие каких-либо специальных оптимизаций. В качестве исходных данных были взяты графы социальных сетей Twitter и Friendster и сгенерированный RMAT граф. Цель работы — исследовать работу GraphX относительно таких архитектурных решений процессоров POWER8 как неоднородная память (NUMA) и одновременная многопоточность (SMT). В отличие от многих других работ, исследующих эффективность Apache Spark с NUMA авторы производят запуске не на одном единственном узле, а на кластере из нескольких машин. Для тестирования были взяты 3 графовых алгоритма: поиск в ширину (Breadth-First Search — BFS), PageRank (PR), поиск компонент связности графа (Connected Components — CC). В режиме запуска “с использованием NUMA” каждому пулу потоков Apache Spark (такой пул называется Spark Worker) выделяется определенный набор процессоров. Потоки могут работать только на этих процессорах и использовать только их память. В обычном режиме поток не привязан к конкретному процессору. Он может

начать выполнение на одном, а потом — быть переданным на исполнение другому процессору. Эксперимент показал, что режим с выделением групп процессоров более эффективен. На алгоритме BFS ускорение составило 13,9% от времени выполнения того же самого алгоритма на тех же данных но в обычном режиме, время выполнения алгоритм поиска общей компоненты (CC) с использованием NUMA уменьшилось на 25,7%, наибольшее улучшение производительности было достигнуто для алгоритма Page Rank — 36,7%. В статье также представлены результаты профилировки использования памяти, диска и сети.

Еще одна работа достойная внимания ученых Университета Каталонии [7] содержит всестороннее исследование производительности всех компонентов Apache Spark особое внимание уделяя тому как возможности современных архитектурных решений в серверах влияют на производительность фреймворка. Было рассмотрено влияние следующих опций: NUMA, Hyper-threading, процессора с архитектурой Ivy Bridge (в частности влияние многоядерной архитектуры процессора). Полный список вопросов рассматриваемых в работе:

- Отличается ли производительность для пакетной и потоковой обработки данных?
- Как скорость передачи данных влияет на производительность вычислений в Spark?
- Как использование NUMA для размещения данных в памяти и организации вычислений влияет на производительность Spark?
- Дает ли прирост производительности использование одновременной многопоточности (SMT)?
- Существуют ли в современных серверах блоки аппаратной предвыборки инструкций данных из оперативной памяти (prefetchers) повы-

шающие эффективность работы Spark?

- Обработка потоковых данных в Spark начинает происходить с задержкой, когда интенсивность поступающего потока данных составляет больше 80% от пропускной способности канала.
- Эффективнее использовать несколько Spark executor (процессов) выделяя каждому небольшое количество памяти или один с большим объемом памяти?

Основные выводы:

- Программы на Spark плохо масштабируются на многоядерной архитектуре из-за частого обращения к DRAM (динамическая память с произвольным доступом) в следствие чего происходит неравномерная загрузка потоков выполнения и, как следствие, неоптимальное время выполнения задачи.
- Производительность Spark уменьшается с увеличением объема входных данных из-за больших затрат на сборку мусора.
- Исполнение программ на Spark в несколько потоков с небольшим объемом выделенной памяти до 30% эффективнее, чем исполнение в один поток с большим объемом выделенной памяти.
- Использование режима Hyper-Threading уменьшает задержки при работе с памятью до 50%.
- Запуск задач с локализацией по NUMA узлам уменьшает время выполнения задач в среднем на 10%.

3 Постановка задачи

В обзоре литературы представлены как всестороннее тестирование фреймворка Apache Spark, так и некоторых отдельно взятых его библиотек. Однако, в работах тестирование проводилось на специально сконфигурированных hadoop-серверах. Для использования Apache Spark в научных целях не всегда есть возможность использовать специальный кластер вместе с тем интерес к технологии стремительно растет. Обычно задачу приходится или запускать локально на отдельно стоящем рабочем узле или на одном из узлов вычислительного кластера через планировщик. Планировщик здесь — система, управляющая выделением ресурсов пользователям кластера. Ресурсы сформированы заранее и настроить их под себя пользователь не может.

Цель работы: исследовать особенности работы Apache Spark на системах с неоднородной памятью, с нестандартная конфигурацией.

Задачи:

1. Предложить способы повышения производительности.

2. Провести тестирование на различных архитектурах.

В качестве тестовой была взята задача подсчета слов Wordcount. При выборе набора данных для тестирования я обратилась к используемым в релевантных исследованиях бенчмаркам. На текущий момент наиболее всеобъемлющим бенчмарком для тестирования стека BigData приложений считается поддерживаемый Китайской академией наук BigDataBench [8]. Для выбранной задачи была использована тестовая версия сайта Wikipedia, содержащая 4 300 000 статей на английском языке. Объем датасета порядка 40 гигабайт. Для тестирования была взята часть датасета объемом 10

гигабайт.

Тестирование проводилось на 2 разных машинах: узел с Intel Xeon Phi KNL в МСЦ РАН и узел wombat-3 в Вычислительном Центре СПбГУ. Конфигурации узлов представлены в соответствующих разделах работы.

Для настройки конфигурации были использованы следующие параметры Apache Spark:

- `spark.driver.memory` — количество памяти, используемое основным процессом Spark;
- `spark.offHeap.memory` — количество памяти, используемой вне кучи Java;
- `spark.worker.memory` — количество памяти, используемое вычислительными процессами Java.

4 Архитектура памяти Apache Spark

В Apache Spark выделяются 3 вида памяти:

1. Reserved memory — это часть памяти, которая резервируется для работы операционной системы и не может быть использована для нужд Apache Spark.

2. User memory — это часть памяти, которая остается после выделения Spark Memory. Здесь пользователь может хранить свои собственные структуры данных, используемые во время вычислений.

3. Spark memory (делится на storage и execution memory) — это память которую непосредственно использует Apache Spark. Ее размер регулируется параметром spark.memory.fraction и вычисляется как (“Java Heap” − “Reserved Memory”) $*$ spark.memory.fraction. В storage memory хранятся закешированные данные на случай перезапуска приложения и значения общих переменных, доступных всем вычислительным потокам. В execution memory хранятся служебные объекты, необходимые во время выполнения заданий (например данные буфера используемые во время фазы shuffle, сортировки данных, агрегирующих операций, операций join).

Следующие классы играют большую роль в процессах управления памятью в Apache Spark.

TaskMemoryManager управляет выделением памяти для выполнения заданий. Задание (task) один из примитивов Apache Spark, выполняющий действие над одной частью (partition) массива данных. Этот класс хранит таблицу выделенных блоков памяти.

MemoryConsumer — абстрактный класс, определяющий операции выделения и освобождения памяти, а также сбрасывания данных на диск во время фазы shuffle.

MemoryMode — определяет режим использования памяти — on-heap или off-heap.

MemoryBlock — является результатом работы функции allocate классов — аллокаторов памяти. Создает блок памяти определенного размера, способен сообщить размер блока, заполнить выделенный блок набором байт с определенным значением.

MemoryAllocator — интерфейс выделения и освобождения памяти, имеющий две реализации: HeapMemoryAllocator для выделения и освобождения памяти внутри JVM и UnsafeMemoryAllocator для выделения и освобождения off-heap памяти. В этих классах происходит непосредственный вызов функций работы с памятью.

Как видно из описания классов, память может быть выделена в памяти Java процесса (on-heap/Java heap) или непосредственно в оперативной памяти (off-heap). Выделение off-heap памяти происходит через функцию allocateMemory класса sun.misc.Unsafe. Выделение on-heap памяти происходит путем создания массива типа long внутри кучи (heap) Java Virtual Machine. Для каждого способа выделения памяти (on-heap, off-heap) в Spark реализован свой Memory Allocator.

Механизм выделения памяти в Spark довольно сильно разветвлен и, как оказалось, через указанные классы-аллокаторы (Memory Allocator) выделяется не вся память, а только часть execution.

Возможность выделения памяти вне кучи Java была добавлена в рамках реализации проекта Tungsten. Проект Tungsten реализуется компанией Databrics с 2015 года и предназначен в первую очередь для компонента Spark SQL, он предполагает как введение нового API для структур Dataset и Dataframe, так и оптимизацию работы с памятью. В частности предла-

гается реализовать более эффективное управление памятью JVM, а также ввести свои кэш-алгоритмы и структуры данных, способные использовать преимущества многоуровневой памяти.

Введение возможности использования off-heap памяти в Spark было призвано решить следующие проблемы:

- Большие накладные расходы из-за работы сборщика мусора Java. В частности, в исследованиях отмечалось, что при запуске большого числа исполнителей (executor-ов) Spark эффективность падает и связано это как раз с работой сборщика мусора. В то же время Spark знает как перемещаются данные во время вычислений засчет чего фреймворк способен более точно оценить время жизни объектов, чем стандартный алгоритм GC.

- "Раздутое"представление данных. Отказ от использования объектов Java и переход к двоичному представлению данных позволяет хранить обекты более компактно.

- Избыток машинных инструкций. Методы класса Unsafe являются внуренними и компилируются в один системный вызов.

Для использования преимуществ быстрой памяти мной введен дополнительный режим использования памяти — numa. Для режима написана реализация интерфейса MemoryAllocator для выделения и освобождения off-heap памяти на указанном узле. Новая реализация использует для выделения памяти функцию библиотеки jnuma. Jnuma — библиотека, предоставляющая API для работы с NUMA памятью на 64-разрядных ОС Linux из языка Java. Реализованные функции являются обертками над библиотекой libnuma, написанной на языке C. Библиотека позволяет выделять память локально или на нескольких NUMA узлах, показывать расстояние между ними, управлять потоками выполнения между узлами.

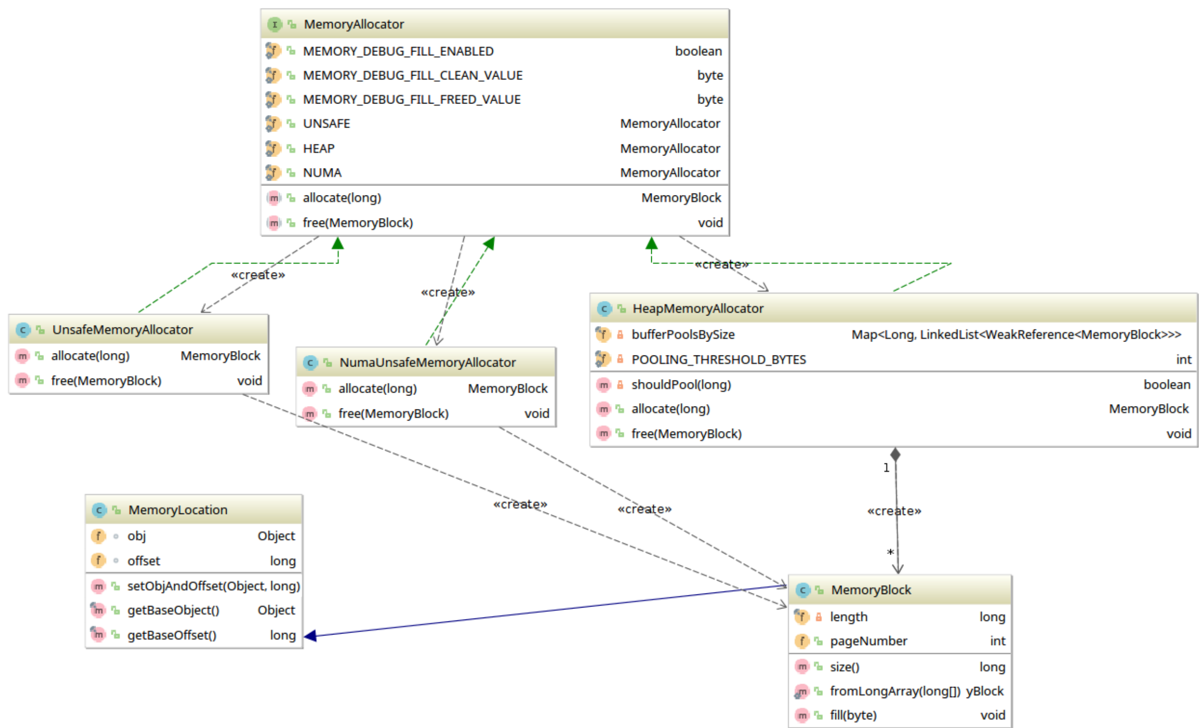

Рис. 1: Схема классов выделения памяти (представлена измененная часть).

Добавлены опции конфигурации:

- `spark.usenuma.bindmemory` принимает значение true или false (по умолчанию второе). указывает, будут ли в конфигурации использоваться настройки NUMA.
- `spark.usenuma.memory.node` определяет память какого узла должна выделяться.
- `spark.usenuma.memory.size` определяет память размер памяти, который может быть использован на указанном узле.

5 Исследование производительности на Intel Xeon Phi KNL

5.1 Выбор методов и тестирование

Сначала рассмотрим тест Apache Spark на Intel Xeon Phi KNL. Конфигурация узла представлена в таблице 2. Этот процессор имеет интересные архитектурные решения: содержит 72 ядра, каждое ядро поддерживает до 4 потоков, в системе это отображается как $72 \cdot 4 = 288$ ядер. Особенностью является наличие модуля сверхбыстрой памяти MCDRAM в добавок к обычной DDR памяти. Архитектура памяти вариативна — MCDRAM память можеть быть использована как L3 кэш, присоединена к DDR памяти или выделена в отдельный модуль. Для эффективной работы с MCDRAM памятью Intel разработала библиотеку memkind, реализующую собственные функции выделения памяти. В API определено несколько вариантов использования памяти, основанных на вариантах возможных архитектурных решений памяти в Intel Xeon Phi KNL. Небольшой опыт использования Xeon Phi KNL и упомянутой библиотеки привел к желанию протестировать успешность работы Apache Spark на этом процессоре и проверить эффективность использования быстрой памяти для выполнения его задач. Тестирование проводилось с использованием режима памяти flat mode. В этом режиме вся быстрая память представлена в виде отдельного NUMA узла, не содержащего вычислительных ядер, обычная память и все вычислительные ресурсы представлены на соседнем узле. Предполагается, что размещение данных, над которыми производятся вычисления, в быстрой

Процессор	Intel Xeon Phi CPU 7290
Количество узлов NUMA node	
node 0	cpus: 0 - 287 size: 98207 MB
node 1	cpus: size: 16384 MB

Схема организации памяти	`node 0 1`
	`0: 10 31`
	`1: 31 10`

Таблица 2: Intel Xeon Phi KNL (МСЦ РАН).

памяти ускорит работу.

Доступ к узлу с Intel Xeon Phi KNL был предоставлен на кластере МСЦ РАН. К сожалению, ограничение на объем данных в пользовательских директории (5 Гб) вынудило уменьшить размер датасета. Размер датасета для теста на KNL составил 2.8 Гб вместо 10. Было рассмотрено 3 разных случая. Результаты представлены в табл. 3.

1. Запуск стандартной версии Apache Spark на узле NUMA 0.

2. Запуск стандартной версии Apache Spark на узле NUMA 0 с указанием использовать память с узла NUMA 1 (MCDRAM). Команда запуска `numactl --membind=1 <команда запуска Spark>`. В этом случае вся память будет выделяться только на узле 1. Если памяти на узле будет недостаточно, то очередная операция выделения памяти завершится с ошибкой, но память на другой ноде выделена не будет.

3. Запуск измененной версии Apache Spark с добавленной собственной функцией выделения памяти. В настройках Spark указывается номер узла с быстрой памятью и класс-аллокатор использует его для выделения execution памяти. Это может несколько уменьшить временной выигрыш от использования быстрой памяти, поскольку часть данных остается в обычной памяти, но зато уменьшает шансы на ошибку во

время выполнения из-за недостатка памяти.

N теста	Версия Spark	MCDRAM	время работы	ускорение
1	исходная	нет	244 сек.	1
2	исходная	да	232 сек.	1.05
3	измененная	да	235 сек.	1.04

Таблица 3: Тест на Intel Xeon Phi KNL.

Для профилировки выделения памяти использовалась утилита *numastat*. Она ведет счетчики выделения страниц памяти на машине. У узла есть 2 основные операции — запрос страницы памяти (на своем или другом узле) и выделение страницы памяти.

- *numa_hit* — увеличивается на 1 на узле 1, если у узла 1 была запрошена страница памяти и эта страница была выделена на запрошенном узле.
- *numa_miss* — увеличивается на 1 на узле 2, если у узла 1 была запрошена страница, но вместо этого она была выделена на узле 2.
- *numa_foreign* — увеличивается на 1 на узле 1, если у узла 1 была запрошена страница, но вместо этого она была выделена на узле 2.
- *interleave_hit* — увеличивается на узле, если выделеная на нем странице соответствует заданной политике.
- *local_node* — увеличивается на 1 на узле, если узел запрашивал страницу памяти у себя и она была успешно выделена.
- *other_node* — увеличивается на 1 на узле, если на узле была выделена страница памяти, но запрос на ее выделение пришел от другого узла.

На снимке экрана 2 представлены значения использования памяти для исполняемого процесса Spark с частотой 5 секунд. Видно, что у узла 1

```
Per-node numastat info (in MBs):
                          Node 0          Node 1           Total
                 --------------- --------------- ---------------
Numa_Hit               167394.78        26811.27       194206.05
Numa_Miss                   0.00            0.00            0.00
Numa_Foreign                0.00            0.00            0.00
Interleave_Hit            132.34          133.59          265.94
Local_Node             167394.78            0.00       167394.78
Other_Node                  0.00        26811.27        26811.27

Per-node numastat info (in MBs):
                          Node 0          Node 1           Total
                 --------------- --------------- ---------------
Numa_Hit               167417.45        26869.95       194287.39
Numa_Miss                   0.00            0.00            0.00
Numa_Foreign                0.00            0.00            0.00
Interleave_Hit            132.34          133.59          265.94
Local_Node             167417.45            0.00       167417.45
Other_Node                  0.00        26869.95        26869.95

Per-node numastat info (in MBs):
                          Node 0          Node 1           Total
                 --------------- --------------- ---------------
Numa_Hit               167428.81        26915.65       194344.46
Numa_Miss                   0.00            0.00            0.00
Numa_Foreign                0.00            0.00            0.00
Interleave_Hit            132.34          133.59          265.94
Local_Node             167428.81            0.00       167428.81
Other_Node                  0.00        26915.65        26915.65
```

Рис. 2: Вывод команды `numastat`.

увеличивается показатель *other_node*, показывающий выделение страниц памяти, запрошенных с другого узла. Узел 0 использует память для служебных целей, поэтому значения показателя *local_node* у него все равно увеличиваются. Но для узла с быстрой памятью этот показатель равен нулю, так как узел не содержит вычислительных ядер и страницы выделяет только по запросу с другого узла.

5.2 Анализ результатов

Как видно из таблицы 3, использование быстрой памяти оправдало себя, хотя и не дало значительного ускорения. Версия Spark с собственным алгоритмом выделения памяти ожидаемо оказалось чуть медленнее, чем просто запуск через numactl с привязкой памяти к одному узлу, так как использование быстрой памяти в этом случае происходит не для всей задачи, а лишь для наиболее затратной ее части — сортировки данных.

Ниже в таблице представлены результаты запуска датасета того же размера на узле wombat-3. Здесь, к сожалению, можно увидеть, что в области использования стека Big Data Xeon Phi сильно проигрывает в производительности обычному серверному узлу. Время выполнения выросло в 6-8 раз. Возможно, он лучше показал бы себя на задачах машинного обучения, где выше итеративность вычислений.

Разница в производительности машин была ожидаема, но не с таким большим разрывом. Однако, не стоит забывать и о других факторах, влияющих на время выполнения задачи. На кластере МСЦ РАН запуск задачи производился через планировщик SLURM. Данные пользователя лежат на одном узле, тогда как выполнение задачи происходит на другом. Время на перенос данных с узла пользователя на узел выполнения на входе и обратно — на выходе может занять значительное время, которое, трудно подсчитать.

6 Исследование производительности на узле wombat-3

6.1 Выбор методов и тестирование

Конфигурация узла представлена в таблице 4. Здесь NUMA узлы равнозначны и в связи с этим выделение памяти на отдельном узле видится бесмысленным, нужны другие методы.

Способы работы с NUMA памятью в Java можно разделить на 2 части: поддержка работы с NUMA внутри самой Java и наличие библиотек, реализующих работу с NUMA для использования в программах, написанных на Java. Библиотека jnuma уже была рассмотрена выше. Что же с поддержкой NUMA в самой JVM?

Поддержка работы с NUMA была впервые добавлена в Java в версии Java SE 6u2 [9] (3 июля 2006 года). Повысить эффективность при работе с NUMA были призваны опции JVM `-XX:+UseParallelGC` и `-XX:+UseNUMA`.

Процессор	Intel Xeon CPU E5-2690 v4
Количество узлов NUMA node	
node 0	cpus: 0 - 13, 28 - 41 size: 128540 MB
node 1	cpus: 14 - 27, 42 - 55 size: 129018 MB

Схема организации памяти	`node 0 1`
	`0: 10 21`
	`1: 21 10`

Таблица 4: Конфигурация узла wombat-3.

Опция `-XX:+UseParallelGC` инициализрует работу сборщика мусора Parallel Scavenger. Этот сборщик мусора был модифицирован таким образом, чтобы учитывать особенности машин с NUMA архитектурой. Раньше его надо было вклюать специально, но сейчас он по умолчанию используется если машина имеет больше одного ядра (то есть почти всегда).

Опция `-XX:+UseNUMA` рекомендует JVM для выделения памяти использовать использовать специальный NUMA аллокатор. Аллокатор делит пространство части кучи JVM где создаются новые объекты на несколько областей, каждая из которых помещается в память конкретного узла. При выделении памяти аллокатор полагает, что поток выделивший объект вероятнее всего будет использовать его снова и снова. Для обеспечения быстрого доступа аллокатор размещает объект в той же области, где находится поток. Области размещения объектов не статичны, их размер может динамически изменяться, увеличиваясь в пользу того узла, на котором интенсивнее используется память. Кроме того гарантируется, что все потоки в среднем имеют равные задеркжи при доступе к объектам старого, нового и постоянного поколений.

До 2012 года в ядре Linux существовал баг, который не позволял использовать опцию UseNUMA — JVM экстренно завершалась с ошибкой. Возможно эта проблема и породила создание библиотеки jnuma, первый релиз которой состоялся 28 ноября 2012 года.

В таблице 5 представлены результаты тестирования. Запуск производился в обычном режиме и с использованием опции UseNUMA. Возможно ранее UseNUMA давала преимущество, но сейчас ее использование не уменьшает время работы программы и даже, наоборот, немного увеличивает. Это видно и в следующем тесте. Здесь 6 использован уменьшенный датасет из предыдущего раздела. На меньших исходных данных еще больше видна разница в использовании UseNUMA. Наконец, таблица 7 представляет результаты теста на том же уменьшенном датасете, но с другим

количеством выделенной памяти. Spark стремится занять всю доступную память, поэтому задачи с одинаковым количеством выделяемой памяти, но разным объемом входных данных будут выполняться с разной скоростью. Для датасета 10 Гб было выделено 20 Гб памяти, соотношение 1:2. Выделим память для меньшего датасета в таком же соотношении. Время выполнения выросло в 1,5 раза.

N теста	UseNUMA	время работы
1	нет	121 сек.
2	да	123 сек.

Таблица 5: Тест на wombat-3. Данные 10 Гб. Выделенная память 20 Гб.

N теста	UseNUMA	время работы
1	нет	30 сек.
2	да	36 сек.

Таблица 6: Тест на wombat-3. Данные 3 Гб. Выделенная память 20 Гб.

N теста	UseNUMA	время работы
1	нет	49 сек.
2	да	53 сек.

Таблица 7: Тест на wombat-3. Данные 3 Гб. Выделенная память 6 Гб.

6.2 Анализ результатов

Использование опции UseNUMA, призванной улучшить работу с неоднородной памятью оказалось неоправданным. Она не только не ускорила

время выполнения задачи, но и наоборот, замедлила. За время ее существования произошел большой скачок в развитие техники и, что неизменно влечет за собой и развитие программного обеспечения. Сейчас разделение на NUMA узлы производится производителями во всех серверах, имеющих больше одного физического процессора. Вместе с этим и современные алгоритмы распределения памяти в JVM вероятно работают лучше, чем когда-то разработанный алгоритм для UseNUMA.

7 Выводы

В ходе работы был произведен запуск Apache Spark на двух конфигурациях узлов для каждого из которых был предложен и обоснован свой вариант оптимизации.

Для Intel Xeon Phi KNL была выбрана оптимальная конфигурация процессора (flat mode, использование быстрой памяти в качестве отдельного узла), добавлены изменения в исходный код Spark, позволяющие настроить его для эффективного выполнения на KNL не путем использования непортируемых команд (numactl), а через параметры программы, которые могут быть заданы как в конфигурационном файле, так и в исходном коде программы-задачи. Предложенный вариант позволил получить небольшое ускорение по сравнению с запуском обычной версии. Однако Xeon Phi KNL сильно проиграл в производительности узлу с более общепринятым Intel Xeon E5 при тех же размерах выделяемой памяти. Частично это можно объяснить тем, что данные и целевой процессор находятся на разных узлах и время на вынужденный перенос данных нельзя подсчитать. Характеристики используемого в МСЦ РАН хранилища также неизвестны, поэтому точно оценить время потраченное процессором на вычисления проблематично.

Для узла wombat-3 было предложено использовать опцию JVM UseNUMA. Это не дало ускорения и даже, напротив, несколько замедлило вычисления. Причем чем меньше общее время выполнения задачи, тем значительнее разница. Такое странное поведение может быть вызвано необновляемостью технологии. Она была предложена в 2006 году, когда многопроцессорные сервера еще не были широко распространены и использование NUMA памяти было в новинку. За это время развитие оборудования повлекло за собой

и обновление технологий и сегодняшний стандартный алгоритм работы с памятью JVM работает лучше, чем ранее специально разработанный.

8 Заключение

В ходе работы было проведено тестирование фреймворка Apache Spark по эффективности работы с неоднородной памятью. Проведен анализ исходного кода, отвечающего за работу с памятью. Добавлена возможность конфигурировать параметры запуска фреймворка на архитектуре с неоднородной памятью через настройки SparkConf. Реализован класс выделяющий память на заданном узле. Получившиеся результаты проанализированы, на основе их даны рекомендации по использованию тех или иных технологий.

9 Глоссарий

NUMA (Non-Unified Memory Access, пер. — неоднородным доступ к памяти) — модель организации памяти многопроцессорной системы когда время доступа к памяти зависит от ее удаленности от обращающегося к ней процессора. В этом случае все процессора в системе делятся на несколько групп, а память компьютера на несколько частей, каждая из которых работает с одной процессорной группой. Одна такая пара процессорной группы и привязанного к ней диапазона адресов общей памяти компьютера считается одним узлом (NUMA node). Собственная кэш-память процессора при этом никуда не исчезает. Неоднородность состоит в том, что работая с общей памятью системы процессор может обратиться как к локальной памяти своего узла, так и к удаленной памяти соседнего узла. Но обращение к памяти соседнего узла идет через шину и скорость доступа будет ниже. В дальнейшем под термином NUMA (память) будет подразумеваться описанная выше архитектура памяти.

NUMA node — один узел NUMA память. Включает в себя процессор или группу процессоров и выделенный им для быстрого доступа диапазон адресов из общей памяти компьютера.

Spark executor — один процесс/поток выполнения задач в Apache Spark.

JVM (Java Virtual Machine) — виртуальная машина Java, основная часть системы выполнения языка Java. JVM выполняет байт-код Java, скомпилированный из исходного кода программы. Компилировать исходный код других языков кроме Java в байт-код JVM также возможно. Один из таких языков — Scala, на котором реализована значительная часть фреймворка Apache Spark.

SMT (Simultaneous Multithreading) — одновременная многопоточность. Форма многопоточности, объединяющая в себе принцип параллельного выполнения инструкций на суперскалярных архитектурах и аппаратную многопоточность.

HPC (High Perfomance Computing) — высокопроизводительные вычисления. В более широком смысле — приложения, вычислительные системы и их окружение, специализирующиеся на вычислениях требующих интенсивного использование процессора и памяти (высокая частота обращений, скорость ответа). Обычно это программы, выполняющие различные научные расчеты.

K-means — алгоритм анализа данных для кластеризации объектов. Алгоритм стремится разбить пространство на заданное количество кластеров таким образом, чтобы сумма квадратов расстояния от центра кластера до его объектов для каждого кластера была возможно минимальной.

Wordcount — стандартная базовая задача в области Big Data, необходимо посчитать сколько раз в тексте встречается каждое слово.

Список иллюстраций

1 Схема классов выделения памяти (представлена измененная часть). 20
2 Вывод команды `numastat`. 24

Список таблиц

1 Сравнение технологий, используемых в Big Data и HPC. . . 7
2 Intel Xeon Phi KNL (МСЦ РАН). 22
3 Тест на Intel Xeon Phi KNL. 23
4 Конфигурация узла wombat-3. 26
5 Тест на wombat-3. Данные 10 Гб. Выделенная память 20 Гб. 28
6 Тест на wombat-3. Данные 3 Гб. Выделенная память 20 Гб. . 28
7 Тест на wombat-3. Данные 3 Гб. Выделенная память 6 Гб. . 28

Список литературы

[1] Hemsoth Nicole. Bringing HPC and Hadoop Under the Same Cluster Umbrella // The Next Platform. 2015.

[2] Optimizing Java and Apache Hadoop for Intel Architecture. URL: https://software.intel.com/sites/default/files/hadoop-and-intel-java-optimization-whitepaper.pdf.

[3] Krishnan Sriram, Tatineni Mahidhar, Baru Chaitanya. myHadoop-Hadoop-on-Demand on Traditional HPC Resources // San Diego Supercomputer Center Technical Report TR-2011-2, University of California, San Diego. 2011.

[4] Spark deployment and performance evaluation on the MareNostrum supercomputer / R. Tous, A. Gounaris, C. Tripiana [и др.] // 2015 IEEE International Conference on Big Data (Big Data). 2015. Oct. С. 299–306.

[5] Gene Resequencing with Myrna on Intel Distribution of Hadoop. URL: https://www.intel.com/content/dam/www/public/us/en/documents/reports/gene-resequencing-with-myrna-distribution-hadoop.pdf.

[6] Performance Analysis of Spark/GraphX on POWER8 Cluster / Xinyu Que, Lars Schneidenbach, Fabio Checconi [и др.] // International Conference on High Performance Computing / Springer. 2016. С. 268–285.

[7] Architectural impact on performance of in-memory data analytics: apache spark case study / Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov [и др.] // arXiv preprint arXiv:1604.08484. 2016.

[8] Bigdatabench: A big data benchmark suite from internet services /

Lei Wang, Jianfeng Zhan, Chunjie Luo [и др.] // High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on / IEEE. 2014. C. 488–499.

[9] Java HotSpot™ Virtual Machine Performance Enhancements. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html.