САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Кафедра компьютерного моделирования и многопроцессорных систем

Выпускная квалификационная работа магистра

Однородный программный интерфейс для параллельных вычислений на кластере

Петряков Иван Владимирович

Направление 02.04.02
«Фундаментальная информатика и информационные технологии»
Магистерская Программа «Распределенные вычислительные технологии»

Научный руководитель
к.ф.-м.н. Ганкевич И. Г.

Санкт-Петербург
2021

1 Оглавление

2 Введение

Текущие исследования по созданию интерфейса программирования высокого уровня для суперкомпьютеров и кластеров направлены на то, чтобы

- изменить существующий функциональный язык для параллельного выполнения программы, написанной на этом языке, в кластере;
- создать движок, позволяющий выполнять направленный ациклический граф задач (программ или скриптов) на кластере с учетом их зависимостей.

Преимущество первого подхода заключается в том, что если у вас уже есть последовательная программа, написанная на функциональном языке, вы можете выполнить ее на кластере с использованием либо компилятора, который генерирует параллельный код, либо библиотеки, которая предоставляет те же функциональные формы (например, map, reduce), которые реализованы для параллельного выполнения на нескольких узлах кластера. Однако этот подход представляет некоторые сложности: обработка спекулятивного выполнения различных ветвей кода и устойчивость к сбоям узлов кластера. Вероятно, главный недостаток этого подхода заключается в том, что на функциональных языках написано не так много высокопроизводительных приложений: большинство из них написано на низкоуровневых императивных языках по соображениям эффективности, и этот подход не предоставляет средств для их выполнения в кластере.

Преимущество второго метода по сравнению с подходом на функциональном языке состоит в том, что он позволяет выполнять произвольные программы и сценарии в кластере и определять информационные зависимости между ними. К сожалению, большинство механизмов рабочих процессов используют XML для определения задач, их аргументов и зависимостей. Этот подход неэффективен, поскольку XML не

является языком программирования, и добавление тегов, представляющих циклы, условные выражения и другие важные конструкции потока управления, приводит к созданию языка сценариев с синтаксисом языка разметки - возможно, наиболее неинтуитивно понятным и многословным способом написания программ. Несмотря на эти недостатки, существуют реализации потоков задач, которые популярны и достаточно развиты, чтобы их можно было использовать для решения реальных проблем.

Таким образом, можно сказать, что функциональный подход не является достаточно высокоуровневым, чтобы его можно было использовать для написания сценариев, выполняющих существующие программы в сложном потоке задач, а потоки задач слишком высокоуровневы для написания сценариев общего назначения. Причина отсутствия промежуточного подхода заключается в том, что планировщики пакетных заданий предоставляют интерфейс для выделения узлов кластера и запуска на них любого исполняемого файла, но интерфейс для написания параллельных программ (MPI) - это просто библиотека, которая динамически связывается с исполняемым файлом и не предоставляется планировщиком. Функциональный подход часто основан на использовании библиотеки MPI, а потоки задач основаны на интерфейсе планировщика пакетных заданий. Это создает разрыв между технологиями, который не позволяет создать универсальный и унифицированный интерфейс для выполнения вычислений в кластере, и заставляет выбрать один из двух подходов.

В то же время планировщики заданий, которые используются при анализе больших данных, такие как YARN [21], не имеют этой проблемы, поскольку они предоставляют низкоуровневый интерфейс на основе Java для запуска приложений. Различные среды программирования, такие как Apache Hadoop [1] и Apache Storm [2], построены поверх этого интерфейса, чтобы обеспечить интерфейс высокого уровня для написания определенных видов программ, таких как пакетная обработка или работа в реальном времени.

Существуют интерфейсы более высокого уровня, такие как Oozie [11]. Эта иерархическая архитектура позволяет выбрать правильный уровень абстракции для программы и представляет собой единый интерфейс для запуска приложений на кластере.

Существует множество фреймворков и языков программирования для параллельных и распределенных вычислений [17, 19, 22, 23], которые успешно применяются как в промышленности, так и в академических кругах, однако все они изолированы и самодостаточны. Основная причина отсутствия общего знаменателя между этими фреймворками и языками заключается в том, что нет протокола или низкоуровневого языка для распределенных вычислений. Для последовательных вычислений у нас есть байт-код (например, LLVM [13], байт-код Java, байт-код Guile), который используется в качестве промежуточного, переносимого и универсального представления программы, написанной на любом языке; также у нас есть ассемблер, который не является переносимым, но все же является популярным промежуточным представлением.

Почему общий низкоуровневый язык существует для последовательных вычислений, но не существует для параллельных и распределенных вычислений? Одна из причин, которая относится как к распределенным, так и к параллельным вычислениям, заключается в том, что люди все еще думают о программах как о последовательности шагов - так же, как люди сами выполняют сложные задачи. Императивные языки, на которых программы записываются в виде последовательности шагов, по-прежнему преобладают в промышленности и академических кругах; это контрастирует с непопулярными функциональными языками, на которых программы написаны как композиции функций без подразумеваемого порядка вычислений. Другая причина, которая относится к распределенным вычислениям, заключается в том, что эти вычисления по своей сути ненадежны и не существует универсального подхода для обработки выхода

из строя узлов кластера. Хотя императивные языки позволяют писать более эффективные программы, они не обеспечивают защиты от взаимных блокировок и не гарантируют отказоустойчивость. Кроме того, их намного сложнее писать, так как человеку приходится работать с изменяемым состоянием (локальные и глобальные переменные, объекты и т. д.), и трудно держать это состояние в голове при написании кода. Функциональные языки минимизируют использование изменяемого состояния, обеспечивают частичную защиту от взаимных блокировок (при условии, что программист не использует блокировки вручную) и могут быть изменены для обеспечения отказоустойчивости. С точки зрения автора, люди понимают потенциал функциональных языков, но еще не осознали этот потенциал, чтобы получить все их преимущества; люди осознали весь потенциал императивных языков, но не знают, как избавиться от их недостатков.

Одна важная особенность, которой не хватает в байт-коде и ассемблере, - это способность взаимодействовать между параллельными процессами. Это взаимодействие является общим определяющим фактором, на котором могут быть построены все фреймворки и языки для параллельных и распределенных вычислений, однако универсального низкоуровневого протокола или языка, описывающего общение, не существует.

Распределенные вычислительные системы представляются как множество индивидуальных сущностей (узлов), которые соединены между собой с помощью специализированных программ: в высокопроизводительных вычислениях — это планировщики пакетных задач, в области больших данных — это планировщики задач, тесно связанные с параллельной файловой системой, в облачных вычислениях — системы управления контейнерами приложений и виртуальными машинами. Несмотря на эффективность этих программ в их предметных областях, они не являются универсальными: вместо того чтобы предоставить пользователю

программный интерфейс для проведения вычислений общего назначения на распределенной системе, они либо предоставляют ограниченный потребностями конкретной предметной области интерфейс (в случае с большими данными и облачными вычислениями), либо интерфейс для параллельного запуска программы на всех узлах и коммуникации параллельных процессов между собой, тесно связанный с конфигурацией системы (в случае высокопроизводительных вычислений).

Основным сдерживающим фактором для создания универсального интерфейса, который бы позволял программировать распределенные системы без непосредственной привязки к количеству узлов, их физическим характеристикам (количеству параллельных процессов, сетевым адресам и т.п.), является свойственная им ненадежность. Действительно, при рассмотрении одного вычислительного узла, его выход из строя является критической ситуацией, которую нельзя разрешить на уровне программного обеспечения, из-за чего все программы, предназначенные для запуска на одном узле, написаны в предположении, что этот узел абсолютно надежен. В то же время, в распределенной системе достаточно большого размера выход из строя одного из узлов является нормальной ситуацией, которая не должна влиять на ее работоспособность в целом. В современных распределенных системах решены лишь частные случаи данной проблемы: в высокопроизводительных вычислениях используются контрольные точки восстановления, в больших данных надежность обеспечивается файловой системой, а в облачных вычислениях, как правило, сетевым хранилищем. Разработчики этих систем как будто бы хотят уйти от обязанности обеспечения отказоустойчивости, переложив ее на распределенную систему хранения данных, которая вычислениями не занимается. Такой подход вряд ли можно назвать эффективным: контрольные точки восстановления работают не для каждой программы и даже для подходящих программ всегда есть шанс ошибки при восстановлении, сохранение всех промежуточных

результатов вычислений в параллельных файловых системах невозможно сделать быстро, а сетевые хранилища на деле являются просто узлами с большим количеством дисков, выход из строя которых может повлечь за собой нарушения функционирования всей распределенной системы.

Таким образом, проблема заключается в специализированности механизмов обеспечения отказоустойчивости распределенных систем, а также их примитивности, что является одной из основных причин отсутствия универсальных средств и интерфейсов для программирования таких систем.

3 Постановка задачи

Целью работы является создание высокоуровневого программного интерфейса для проведения вычислений общего назначения на кластере, который абстрагирует программиста от ненадежности узлов кластера, сложностей распределения нагрузки между узлами кластера, а также частично предоставляет программисту автоматический параллелизм.

Основной задачей является создание высокоуровневого программного интерфейса на функциональном языке Guile, который реализован на основе низкоуровневого интерфейса планировщика Subordination [7-9]. Высокоуровневый интерфейс отображает формы языка Guile на управляющие объекты, которые используются низкоуровневым интерфейсом для распределения нагрузки между узлами кластера, предоставления отказоустойчивости и параллелизма.

Второй задачей является проверка эффективности и высокоуровневого интерфейса проверяется с помощью синтетических тестов производительности, а также путем переписывания реального приложения с использованием этого интерфейса.

Глобальной целью является вывод программных интерфейсов для параллельных и распределенных вычислений на уровень, на котором они будут создаваться так же, как и последовательные программы, т. е.

распределение нагрузки и отказоустойчивость будут автоматически предоставляться каждой программе, использущей данный интерфейс.

4 Обзор литературы

Чтобы писать параллельные и распределенные программы так же, как мы пишем последовательные программы, нам понадобятся следующие компоненты.

- Низкоуровневый язык, который выполняет роль промежуточного переносимого представления кода и данных и включает средства декомпозиции кода и данных на части, которые могут быть вычислены параллельно. Ближайший последовательный аналог — LLVM.
- Планировщик кластера, который выполняет параллельные и распределенные приложения и использует низкоуровневый язык для реализации связи между этими приложениями, запущенными на разных узлах кластера. Ближайшим одноузловым аналогом является ядро операционной системы, выполняющее пользовательские процессы.
- Интерфейс высокого уровняк для существующих популярных языков программирования в виде фреймворка или библиотеки, который основан на низкоуровневом языке. Этот интерфейс использует планировщик кластера, если он доступен и приложению требуется параллелизм узлов, в противном случае код выполняется на локальном узле, и параллелизм ограничивается параллелизмом узла. Ближайшим одноузловым аналогом является библиотека C, которая обеспечивает интерфейс для системных вызовов ядра операционной системы.

При таком подходе низкоуровневый язык отвечает за обеспечение параллелизма и отказоустойчивости приложений, планировщик кластера обеспечивает прозрачное выполнение приложений на нескольких узлах кластера, а высокоуровневый интерфейс отображает базовую систему на высокоуровневый язык, чтобы упростить работу для разработчиков приложений.

Технологии высокопроизводительных вычислений имеют ту же трехкомпонентную структуру: библиотека передачи сообщений (MPI) считается низкоуровневым языком параллельных вычислений, планировщики пакетных заданий используются для распределения ресурсов, а высокоуровневый интерфейс - это некоторая библиотека, построенная на основе MPI; однако обязанности компонентов четко не разделены, и иерархическая структура не соблюдается. MPI обеспечивает средства связи между параллельными процессами, но не предоставляет средств декомпозиции данных и гарантий отказоустойчивости: декомпозиция данных выполняется либо на языке высокого уровня, либо вручную, а отказоустойчивость обеспечивается планировщиком пакетных заданий. Планировщики пакетных заданий предоставляют средства для распределения ресурсов (узлов кластера, ядер процессора, памяти и т. д.) и запуска параллельных процессов MPI, но не контролируют сообщения, которые отправляются между этими процессами, и не контролируют фактическое количество ресурсов, используемых программой (все ресурсы принадлежат исключительно программе, и программа решает, как их использовать), то есть планировщики кластеров и программы MPI не взаимодействуют друг с другом после запуска параллельных процессов. Следовательно, высокоуровневый интерфейс также отделен от планировщика. Хотя высокоуровневый интерфейс построен поверх низкоуровневого интерфейса, планировщик пакетных заданий полностью не интегрирован ни с одним из них: общекластерный аналог ядра операционной системы не контролирует обмен данными между приложениями, которые выполняются на кластере, но вместо этого используется как распределитель ресурсов.

Ситуация в более новых технологиях больших данных иная: есть те же три компонента с иерархической структурой, но язык низкого уровня интегрирован в планировщик. Существует кластерный планировщик YARN [14] с API, который используется в качестве низкоуровневого языка для

параллельных и распределенных вычислений, а также существует множество высокоуровневых библиотек, построенных на основе YARN [1, 2, 11, 20]. Планировщик имеет больший контроль над выполнением заданий, поскольку задания разбиваются на подзадачи, а выполнение подзадач контролируется планировщиком. К сожалению, из-за отсутствия общего низкоуровневого языка все высокоуровневые фреймворки, построенные на основе YARN API, используют свой собственный протокол для связи, перекладывают ответственность за обеспечение отказоустойчивости на планировщик и перекладывают ответственность за декомпозицию данных на более высокий уровень.

Подводя итог, можно сказать, что современные технологии для параллельных и распределенных вычислений можно разделить на три класса: языки низкого уровня, планировщики кластеров и интерфейсы высокого уровня; однако ответственность каждого класса четко не разделена разработчиками этих технологий. Хотя структура компонентов напоминает ядро операционной системы и интерфейс приложения, компоненты иногда не построены друг над другом, а интегрированы по горизонтали, и в результате сложность параллельных и распределенных вычислений иногда видна на высоких уровнях абстракции.

В данной работе описывается интерфейс высокого уровня, построенные на основе низкоуровневого интерфейса кластерного планировщика Subordination. Этот интерфейс полностью скрывает особенности реализации низкоуровневого интерфейса, а также по умолчанию предоставляет параллелизм для программ.

5 Архитектура распределенной системы

5.1 Управляющие объекты

Для создания низкоуровневого языка для параллельных и распределенных вычислений, Subordination заимствует знакомые функции из последовательных низкоуровневых языков и дополняет их асинхронными вызовами функций и возможностью чтения и записи кадров стека вызовов.

На ассемблере и LLVM программа записывается в императивном стиле в виде серии инструкций процессора. Переменные хранятся либо в стеке (особая область основной памяти компьютера), либо в регистрах процессора. Логические части программы представлены функциями. Вызов функции помещает все аргументы функции в стек, а затем переходит к адресу функции. Когда функция завершается, результат вычисления записывается в регистр процессора, а поток управления возвращается в вызывающую функцию. Когда основная функция завершается, завершается и программа.

Есть две проблемы с ассемблером, которые необходимо решить, чтобы использовать его в параллельных и распределенных вычислениях. Во-первых, содержимое стека нельзя копировать между узлами кластера или сохранять в файл и читать из него, потому что они часто содержат указатели на блоки памяти, которые могут быть не доступны на другом узле кластера или в процессе, считывающем стек из файла. Во-вторых, в этом языке нет естественного способа выразить параллелизм: все вызовы функций синхронны, а все инструкции выполняются в указанном порядке. Для решения этих проблем Subordination использует объектно-ориентированные методы.

Мы представляем каждый кадр стека объектом: локальные переменные становятся полями объекта, и каждый вызов функции раскладывается на код, который предшествует вызову функции; код, выполняющий вызов функции,

и код, следующий за ним. Код, который идет до вызова, помещается в метод «act» объекта, а после этого кода создается новый объект для асинхронного вызова функции. Код, который идет после вызова, помещается в метод «react» объекта, и этот код вызывается асинхронно при возврате вызова функции (этот метод принимает соответствующий объект в качестве аргумента). У объекта также есть методы «read» и «write», которые используются для чтения и записи его полей в файл и из файла или для копирования объекта на другой узел кластера. В этой модели каждый объект содержит достаточно информации для выполнения соответствующего вызова функции, и мы можем выполнять эти вызовы в любом порядке. Кроме того, объект является самодостаточным, и мы можем попросить другой узел кластера выполнить вызов функции или сохранить объект на диск для выполнения вызова, когда пользователь хочет возобновить вычисления (например, после обновления и перезагрузки компьютера).

Вызов функций выполняется асинхронно с помощью пула потоков. Каждый пул потоков состоит из очереди объектов и массива потоков. Когда объект помещается в очередь, один из потоков извлекает его и вызывает метод «act» или «react» в зависимости от состояния объекта. Есть два состояния, которые контролируются программистом: в состоянии «upstream» вызывается метод «act», в состоянии «downstream» вызывается метод «react» родительского объекта с текущим объектом в качестве аргумента. Когда в состоянии «downstram» у объекта нет родителя, программа завершается.

Эти объекты являются отличительной особенностью от других инструментов параллельного программирования. Возможность указывать иерархические зависимости между объектами, обрабатываемыми параллельно делает очевидным механизм устойчивости к сбоям. Если вызов какого-либо метода объекта завершается неудачей, то установленные иерархические связи легко позволяют выделить родительский объект. В таком случае, этот объект становится ответственным за повторное

выполнение метода подчиненного объекта, завершившегося неудачей, на оставшихся узлах кластера. Чтобы повторно выполнить метод объекта, у которого нет родительского, создается его копия и отправляется на другой узел.

Основная цель предлагаемого подхода - упростить разработку приложений для распределенной пакетной обработки данных и промежуточного программного обеспечения. Идея состоит в том, чтобы обеспечить устойчивость к сбоям на самом низком уровне. Реализация разделена на два уровня: нижний уровень состоит из процедур и классов для приложений с одним узлом (без сетевых взаимодействий), а верхний уровень для приложений, работающих на произвольном количестве узлов. Существует два типа тесно связанных сущностей — управляющие объекты (или ядра для краткости) и конвейеры, которые составляют основу программы.

Ядра реализуют логику потока управления в своих методах «act» и «react» и хранят состояние текущей ветви потока управления. И логика, и состояние реализуются программистом. В методе «act» некоторая функция либо вычисляется напрямую, либо распадается на вызовы вложенных функций (представленных набором подчиненных ядер), которые впоследствии отправляются в конвейер. В методе «react» подчиненные ядра, возвращенные из конвейера, обрабатываются их родителем. Вызовы методов «act» и «react» являются асинхронными и выполняются в потоках, связанных с конвейером. Для каждого ядра «act» вызывается только один раз, а для нескольких ядер вызовы выполняются параллельно друг другу, тогда как метод «react» вызывается один раз для каждого подчиненного ядра, и все вызовы выполняются в одном потоке для предотвращения состояний гонки.

Конвейеры реализуют асинхронные вызовы методов «act» и «react» и пытаются сделать как можно больше вызовов параллельно, учитывая параллелизм платформы (количество процессорных ядер на узел и

количество узлов в кластере). Конвейер состоит из очереди ядер, которая содержит все подчиненные ядра, отправленные их родителями, и набора потоков, который обрабатывает ядра в соответствии с правилами, описанными в предыдущем абзаце. Для каждого устройства используется отдельный конвейер: существуют конвейеры для параллельной обработки, обработки на основе расписания (периодические и отложенные задачи) и конвейер «прокси» для обработки ядер на других узлах кластера.

В принципе, механизмы ядра и конвейеров повторяют механизм процедур и стеков вызовов с тем преимуществом, что методы ядра вызываются асинхронно и параллельно друг другу (насколько позволяет логика программы). Поля ядра - это локальные переменные стека, метод «act» - это последовательность инструкций процессора перед вызовом вложенной процедуры, а метод «react» - это последовательность инструкций процессора после вызова. Создание и отправка подчиненных ядер на конвейер - это вызов вложенной процедуры. Два метода необходимы, чтобы сделать вызовы асинхронными и заменить активное ожидание завершения подчиненных ядер на пассивное. Конвейеры, в свою очередь, позволяют реализовать пассивное ожидание и вызывать соответствующие методы ядра, анализируя их внутреннее состояние.

Этот низкоуровневый язык можно рассматривать как адаптацию классического стека вызовов функций, но с асинхронными вызовами функций и возможностью чтения и записи кадров стека. Эти различия дают ядрам следующие преимущества

- Ядра определяют зависимости между вызовами функций, но не определяют порядок вычислений. Это дает естественный способ выражения параллельных вычислений на самом низком уровне.
- Ядра могут быть записаны и прочитаны с любого носителя: файлы, сетевые соединения, последовательные соединения и т. д. Это позволяет эффективно реализовать отказоустойчивость, используя

любые существующие методы: для реализации контрольных точек программисту больше не нужно сохранять содержимое памяти каждого параллельного процесса: для перезапуска программы с последнего последовательного шага нужны только поля основного ядра. Однако, с ядрами контрольные точки могут быть заменены простым перезапуском: когда узел, на который было отправлено дочернее ядро, выходит из строя, копию этого ядра можно отправить на другой узел без остановки программы и без дополнительной настройки со стороны программиста.

- Наконец, ядра достаточно просты, чтобы их можно было использовать в качестве промежуточного представления для языков высокого уровня: либо через модификацию компилятора, либо через обертку для каждой библиотеки, которая напрямую вызывает низкоуровневую реализацию. Ядра не могут заменить LLVM или ассемблер, потому что их уровень абстракции выше, поэтому модификация компилятора возможна только для языков, которые используют промежуточное представление высокого уровня (например, LISP-подобные языки и чисто функциональные языки, которые имеют естественный способ выражения параллелизма с помощью параллельного вычисления аргументов функций).

5.2 Планировщик

Планировщик использует ядро как единицу планирования: распределяет ядра по ядрам процессора и узлам кластера для извлечения параллелизма из вычислительной системы. По сравнению с обычными планировщиками пакетных заданий этот планировщик работает на более низком уровне: ядро часто представляет собой единицу работы, которая выполняется последовательно и меньше по времени, чем длительное параллельное пакетное задание. Также, в отличие от пакетного задания, ядро

может быть отправлено на любой узел кластера, что позволяет использовать очень сложные алгоритмы распределения нагрузки, учитывающие текущее и предыдущие состояния всех узлов кластера (средняя нагрузка, количество обработанных ядер. в секунду, объем памяти, дисковое пространство и т. д.).

Планировщик ядра реализован как набор очередей, в которые ядра помещаются некоторым потоком и из которых они извлекаются пулом потоков, присоединенным к очереди. Для каждого устройства узла кластера существуют отдельные очереди:

- очередь процессора с одним потоком на каждое ядро процессора;
- очередь таймера с одним потоком, которая позволяет запускать ядра в заданные моменты времени;
- дисковая очередь с однопоточным вводом / выводом на диск;
- очередь сетевого интерфейса с одним потоком для каждой сетевой карты;
- очередь процессов с одним потоком для связи с дочерними и родительскими процессами.

В программе, выполняемой планировщиком, доступ к очередям осуществляется как к глобальным переменным. Очереди, которые не нужны конкретной программе, можно отключить, чтобы не использовать память и другие ресурсы операционной системы.

Используя ядра, можно написать любую программу. Первой программой, которая была написана с их помощью, стал планировщик кластера, который использует ядра для реализации своей внутренней логики и для запуска приложений, охватывающих несколько узлов кластера. Одноузловая версия планировщика состоит из пула потоков, прикрепленного к очереди ядер. Программа начинается с помещения первого (или основного) ядра в очередь и заканчивается, когда основное ядро меняет свое состояние на нисходящее и помещает себя в очередь. Количество потоков в пуле равно количеству ядер процессора, но может быть установлено вручную, чтобы

ограничить степень параллелизма. Кластерная версия планировщика более сложна и использует ядра для реализации своей логики.

Планировщик кластера запускается в отдельном процессе на каждом узле кластера, и процессы взаимодействуют друг с другом с помощью ядер: процесс на узле *A* записывает некоторое ядро в сетевое соединение с узлом *B*, а процесс на узле *B* читает ядро и выполняет с ним полезные операции. Здесь ядра используются как сообщения, а не фреймы стека: ядро, которое всегда находится в узле *A*, создает дочернее ядро и отправляет его ядру, которое всегда находится в узле *B*, в качестве сообщения. Чтобы реализовать эту логику, было добавлено состояние «point-to-point» и поле, являющееся идентификатором целевого ядра. В дополнение к этому были добавлены поля адреса источника и назначения, чтобы иметь возможность маршрутизировать ядро на целевой узел кластера и возвращать его обратно на узел источника: кортеж *(родительское ядро, адрес источника)* однозначно определяет местоположение родительского ядра, а кортеж *(целевое ядро, адрес назначения)* однозначно определяет местоположение целевого ядра. Первый кортеж также используется нижестоящими ядрами, которые возвращаются к своим родителям, но второй кортеж используется только ядрами в состоянии «point-to-point».

Планировщик кластера выполняет несколько задач:

- запускает приложения в дочерних процессах,
- маршрутизирует ядра между процессами приложений, запущенными на разных узлах кластера,
- ведет список доступных узлов кластера.

Чтобы реализовать эти задачи, были созданы очереди ядер и пул потоков для каждой задачи, с которой должен иметь дело планировщик:

- очередь таймера для плановых и периодических задач,
- сетевая очередь для отправки и получения ядер от других узлов кластера,

- очередь процессов для создания дочерних процессов и отправки и получения от них ядер,
- основная очередь для параллельной обработки ядер с использованием нескольких ядер процессора.

Такое разделение задач позволяет производить параллельно передачу и обработку данных: в то время как основная очередь обрабатывает ядра параллельно, очередь процессов и сетевая очереди отправляют и получают другие ядра.

Планировщик кластера запускает приложения в дочерних процессах; Такой подход естественен для UNIX-подобных операционных систем, поскольку родительский процесс полностью контролирует свои дочерние процессы: количество ресурсов может быть ограничено (количество ядер процессора, объем памяти и т. д.) и процесс может быть завершен в любое время. После введения дочерних процессов в планировщик, было добавлено общекластерное поле идентификатора исходного (целевого) приложения, которое однозначно идентифицирует исходное (целевое) приложение, из которого было получено (которому было отправлено) ядро. Также каждое ядро содержит дескриптор приложения, который указывает, как запускать приложение (аргументы командной строки, переменные среды и т. д.), и если соответствующий процесс не запущен, он запускается автоматически. Дочерние процессы необходимы только как средство управления ресурсами: процесс является единицей планирования для ядра операционной системы, но в планировщике кластера дочерний процесс выполняет что-то полезное только тогда, когда ядро (которое является единицей планирования в нашем планировщике) отправляется в соответствующее приложение, и процесс запускается автоматически, если такого приложения нет. Приложение охватывает несколько узлов кластера и может иметь любое количество дочерних процессов (но не более одного процесса на узел). Эти процессы запускаются по запросу и ничего не делают, пока не будет получено ядро.

Такое поведение позволяет реализовать динамический параллелизм: программисту не нужно указывать количество параллельных процессов при запуске приложения, планировщик автоматически их создаст. Для уменьшения потребления памяти процессы, которые долгое время не получали ядро, могут быть завершены (новые процессы все равно будут созданы автоматически, когда соответствующее ядро придет на узел). Также ядра можно отправлять из одного приложения в другое, указав другой дескриптор приложения.

Дочерний процесс взаимодействует со своим родителем, используя оптимизированную очередь процессов. Если соединение с родительским процессом не удается установить, дочерний процесс продолжает выполнение на локальном узле: приложения, написанные с использованием интерфейса планировщика кластера, работают правильно, даже когда планировщик недоступен, но используют локальный узел вместо кластера.

Поскольку у узла может быть несколько дочерних процессов, может возникнуть ситуация, когда все они попытаются использовать все ядра процессора, что приведет к избыточному потреблению ресурсов операционной системы и пониженной производительности. Чтобы решить эту проблему, мы вводим поле «вес», которое сообщает, сколько потоков будет использоваться ядром. По умолчанию для обычных ядер используется один поток, а для внутренних ядер, которые необходимы для функционирования планировщика кластера, - ноль потоков. Очередь процессов отслеживает количество ядер, которые были отправлены дочерним процессам, и ставит в очередь входящие ядра, если общее количество превосходит количество потоков процессора. Кроме того, каждый кластер сообщает эту информацию другим узлам для принятия оптимальных решений по балансировке нагрузки.

Планировщик действует как маршрутизатор для ядер: когда ядро получено от приложения, планировщик анализирует его поля и решает, на

какой узел кластера оно может быть отправлено. Если ядро находится в состоянии «downstream» или в состоянии «point-to-point», то ядро отправляется на узел, где находится целевое ядро; если ядро находится в состоянии «upstream», алгоритм балансировки нагрузки решает, на какой узел отправить ядро. Алгоритм балансировки нагрузки отслеживает общий вес ядер, которые были отправлены на указанный узел, а также получает ту же информацию от узла (в случае, если другие узлы также отправляют туда свои ядра), затем он выбирает узел с наименьшим весом и отправляет ядро к этому узлу. Если все узлы заполнены, ядро остается в очереди до тех пор, пока не станет доступно достаточное количество ядер процессора.

Последняя, но не менее важная обязанность планировщика - формировать и обновлять список узлов кластера и устанавливать постоянные сетевые соединения с соседними узлами. Планировщик кластера делает это автоматически, сканируя сеть с использованием эффективного алгоритма: узлы в сети организованы в искусственную топологию дерева с заданным значением ветвления, и каждый узел пытается связаться с узлами, которые находятся ближе к корню дерева. Такой подход значительно сокращает количество данных, которые необходимо отправить по сети, чтобы найти все узлы кластера: в идеальном случае только одно ядро отправляется и принимается от родительского узла. Алгоритм описан в [9]. После установления соединений все вышестоящие ядра, полученные от дочерних процессов приложений, направляются на соседние узлы в топологии дерева (как родительские, так и дочерние узлы). Это создает проблему, потому что количество узлов, находящихся «за» родительским, обычно отличается от количества узлов, находящихся «за» дочерними. Чтобы решить эту проблему, мы отслеживаем не только общий вес всех ядер соседних узлов, но и общий вес каждого узла в кластере и складываем вес всех узлов, которые находятся за узлом *A*, чтобы вычислить общий вес узла *A* для балансировки нагрузки. Кроме того, мы применяем балансировку нагрузки рекурсивно:

когда ядро достигает узла, алгоритм балансировки нагрузки выполняется еще раз, чтобы решить, можно ли отправить ядро локально или на другой узел кластера (кроме исходного узла). Такой подход решает проблему, и теперь приложения можно запускать не только на корневом узле, но и на любом узле без проблем с балансировкой нагрузки. Этот подход добавляет небольшие накладные расходы, поскольку ядро проходит через промежуточный узел, но если накладные расходы нежелательны, приложение может быть запущено на корневом узле. Обнаружение узлов и обновления состояния узлов реализованы с использованием ядер «point-to-point».

Подводя итог, планировщик кластера использует ядра как единицу планирования и как протокол связи между своими процессами-демонами, запущенными на разных узлах кластера. Процесс-демон действует как посредник между процессами приложений, запущенными на разных узлах кластера, и все ядра приложений проходят через этот процесс. Ядра, которые отправляются через планировщик, более «тяжеловесны», чем локальные: у них больше полей, чем у локальных ядер, а маршрутизация через планировщик приводит к множеству накладных расходов по сравнению с прямым обменом данными. Однако, использование кластерного планировщика значительно упрощает разработку приложений, поскольку разработчику приложения не нужно беспокоиться о сетевом взаимодействии, отказоустойчивости, балансировке нагрузки и искать ответ на вопрос «сколько параллельных процессов достаточно для моего приложения»: теперь этим занимается планировщик. Для максимальной эффективности, а также для встроенных систем приложение может быть напрямую связано с планировщиком, чтобы иметь возможность работать в одном и том же процессе демона, таким образом накладные расходы планировщика минимальны.

5.3 Ручной параллелизм

Подход к написанию параллельных программ с использованием ядер- это не что иное, как старая модель стека вызовов функций, но с асинхронным выполнением каждого вызова вложенной функции и возможностью скопировать стек на другой узел кластера для выполнения функции там. Следовательно, его можно использовать как модель, в которую транслируются существующие функциональные языки программирования. Самый простой способ перевести язык программирования, подобный LISP, в эту модель - выполнить параллельное вычисление всех аргументов процедуры, т. е. транслировать каждый вызов процедуры в программе в родительское ядро, которое создает подчиненное ядро для каждого аргумента процедуры, чтобы вычислять его параллельно с другими подчиненными ядрами и возвращать результат в родительское ядро. Родительское ядро, в свою очередь, выполняет процедуру как обычно с вычисленными аргументами. Хотя этот подход прост, процедуры LISP часто работают с одним аргументом, который является списком. Чтобы реализовать эти процедуры с использованием ядер, мы должны перевести языковые формы, такие как map и fold (на которых основано большинство таких процедур), в ядра, а затем реализовать процедуры с использованием этих форм.

Аналогичный подход используется в Apache Spark, но он также заменяет списки распределенными наборами данных. Предлагаемый подход не имеет наборов данных, но обеспечивает доступ более низкого уровня к параллельным вычислениям с использованием ядер.

```
;; Guile
(map (lambda () ...) lst)
;; Guile с ядрами
(kernel-map '(lambda () ...) lst)
(kernel-react
        ;; накопление результатов
        '(lambda (values acc) (append values acc))
        ;; финальное действие с накоплением
        '(lambda (acc) (pretty-print acc))
        ;; начальное значение
        '())
```

Листинг 1: Реализация map в языке Guile, основанная на ядрах

Для того, чтобы показать жизнеспособность предложенного подхода, были реализованы некоторые формы диалекта Scheme под названием Guile с использованием ядер. Была переписана функция языка Guile map, с помощью низкоуровнего интерфейса. Сначала вызывается процедура kernel-map, чтобы создать дочерние ядра, которые будут применять эту процедуру к каждому элементу списка. Затем вызывается kernel-react, чтобы отправить дочерние ядра в очередь для параллельного выполнения вычислений. Эта функция принимает процедуру, которая накапливает значения, возвращаемые каждым дочерним ядром, в качестве первого аргумента, процедуру, которая применяется к окончательному накопленному значению, в качестве второго аргумента и начального значения аккумулятора.

5.4 Автоматический параллелизм

Низкоуровневый интерфейс и планировщик кластера написаны на языке C++. С точки зрения автора C является слишком низкоуровневым, а Java имеет слишком много накладных расходов для кластерных вычислений, тогда как C++ - это язык программирования расположенный «между» ними. Реализация представляет собой прямое отображение идей, обсужденных в предыдущих разделах об абстракциях C++: ядро - это базовый класс для всех объектов потока управления с общими полями (родительский, целевой и все

остальные) и «act», «react», «read», «write» – виртуальные функции, которые переопределяются в подклассах. Это прямое отображение является естественным для языка со смешанной парадигмой, такого как C++, но функциональные языки могут выиграть от реализации тех же идей в компиляторе или интерпретаторе.

Была сделана эталонная реализация ядер для языка Guile [6]. Guile - это диалект Scheme [20], который, в свою очередь, является диалектом LISP [16]. Отличительной чертой языков, подобных LISP, является гомоиконичность, то есть код и данные представлены древовидной структурой (списки, которые могут содержать другие списки в качестве элементов). Эта функция позволяет выразить параллелизм непосредственно на языке: каждый элемент списка может быть вычислен независимо и может быть отправлен на другие узлы кластера для параллельных вычислений. Для реализации параллелизма был создан интерпретатор Guile, который вычисляет каждый элемент списка параллельно с использованием ядер. На практике это означает, что каждый аргумент вызова процедуры (вызов процедуры также является списком, первым элементом которого является имя процедуры) вычисляется параллельно. Этот интерпретатор может запускать любую существующую программу Guile (при условии, что она не использует явно потоки, блокировки и семафоры), и вывод будет таким же, как и в исходном интерпретаторе, программа будет автоматически использовать узлы кластера для параллельных вычислений, а отказоустойчивость будет автоматически предоставлена кластерным планировщиком. Такой подход является наиболее прозрачным и безопасным способом написания параллельных и распределенных программ с четким разделением зон ответственности: программист заботится о логике приложения, а планировщик кластера заботится о параллелизме, балансировке нагрузки и отказоустойчивости.

Интерпретатор состоит из стандартного цикла read-eval-print, из которого только этап eval использует ядра для параллельных и

распределенных вычислений. Внутри eval используется гибридный подход для параллелизма: ядра используются для асинхронного вычисления аргументов вызовов процедур и аргументов примитива cons, только если эти аргументы содержат вызовы других процедур. Это означает, что все простые аргументы (переменные, символы, другие примитивы и т. д.) вычисляются последовательно без создания дочерних ядер.

Выполнение процедур и примитива «cons» с использованием ядер оказывается достаточно, чтобы сделать «map» параллельной, но процедуру «fold» необходимо переписать, чтобы сделать ее параллельной. Предложенный параллелизм основан на том факте, что аргументы процедуры могут вычисляться параллельно, не влияя на корректность процедуры, однако параллельное вычисление аргументов в «fold» не дает ускорения из-за вложенных вызовов «fold» и процедуры «proc»: следующий рекурсивный вызов «fold» ждет, пока вызов «proc» завершится. Альтернативная процедура «fold-pairwise» не имеет этого недостатка, но корректна только для ассоциативной процедуры «proc», которая не заботится о порядке аргументов (операторы +, * и т. д.). В этой процедуре мы применяем «proc» к последовательным парам элементов из начального списка, после чего рекурсивно вызывается «fold-pairwise» для результирующего списка. Итерация продолжается до тех пор, пока в списке не останется только один элемент, затем мы возвращаем этот элемент как результат вызова процедуры. Эта новая процедура также итеративна, но параллельна внутри каждой итерации. Формы «map» и «fold» были выбраны, чтобы проиллюстрировать автоматический параллелизм, потому что многие другие формы основаны на них [10]. Реализация этих форм показана в листинге 2.

```
(define (map proc lst) ;;”Параллельный map. ”
        (if (null? lst) lst
                (cons (proc (car lst)) (map proc ( cdr lst )))))
(define (fold proc init lst) ;;”Последовательный fold”
        (if (null? lst) init
                (fold proc (proc (car lst) init) (cdr lst))))
(define (do-fold-pairwise proc lst)
        (if (null? lst) '()
                (if (null? (cdr lst)) lst
                (do-fold-pairwise proc
                (cons (proc (car lst) (car (cdr lst)))
                (do-fold-pairwiseproc (cdr (cdr lst))))))))
(define (fold-pairwise proc lst );; ”Параллельный fold . ”
        (car (do-fold-pairwise proc lst)))
```

Листинг 2: Параллельные map и fold в Guile

6 Результаты

Чтобы протестировать введенные концепции, было разработано приложение производящее обработку данных NDBC[1], собираемых метеобуями по всему миру, и восстанавливающее частотно-направленные волновые спектры из этих данных. Данные для каждого спектра хранятся в пяти файлах, по одному файлу для каждого параметра из следующей формулы [24].

$$S(\omega,\theta)=\frac{1}{\pi}\left(\frac{1}{2}+r_i\cos(\theta-\alpha_1)+r_2\sin(2(\theta-\alpha_2))\right)S_0(\omega) \quad (3)$$

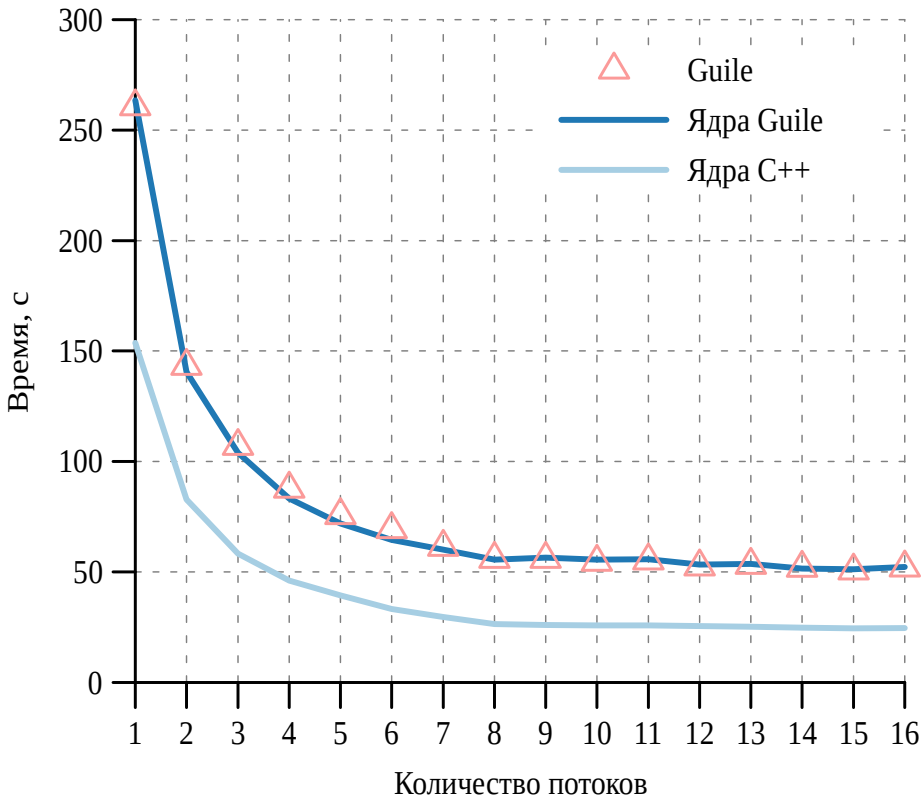


Рисунок 1: Производительность программы, написанной на чистом Guile, адаптированном Guile и C++ на одном узле кластера

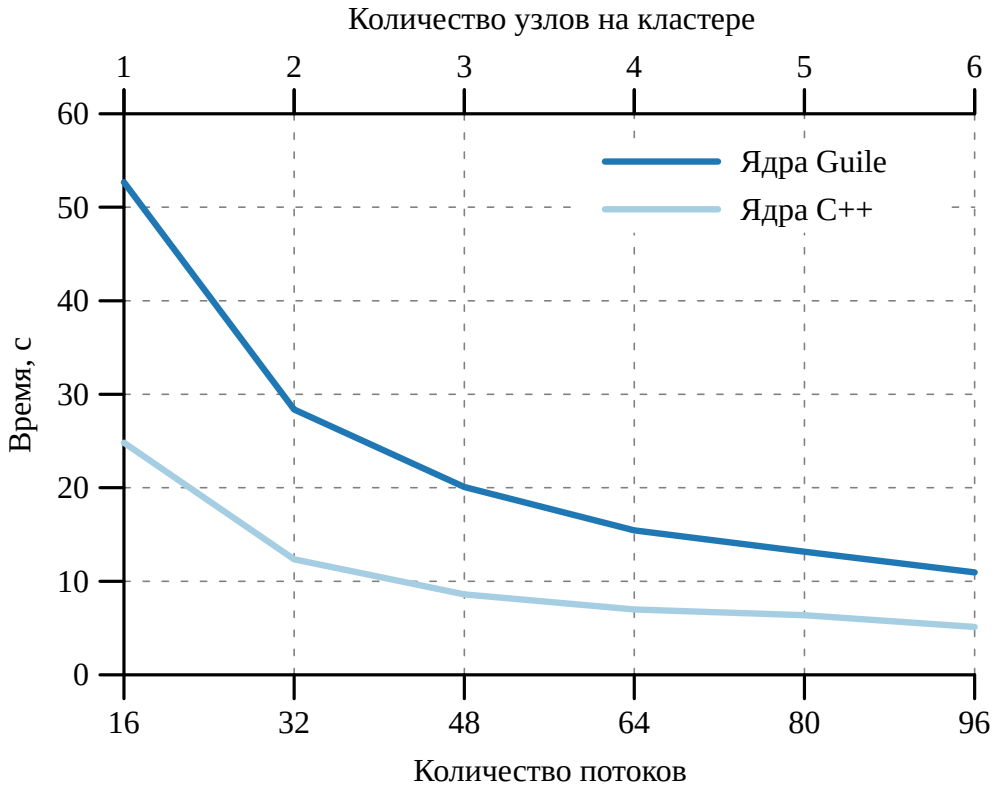
Здесь S_0, r_1,r_2, α_1,α_2 - параметры, ω - частота, θ - направление. Данные не

1 https://www.ndbc.noaa.gov/docs/ndbc_web_data_guide.pdf

согласованы: в пятерке файлов могут быть отсутствующие файлы, а в файле отсутствующие записи. Каждый файл представляет собой набор записей для определенного параметра и определенной отметки времени. Чтобы восстановить спектр, мы должны сначала очистить данные: найти файлы, соответствующие каждому году и станции, которые хранят различные переменные спектра, найти все записи в каждом файле, соответствующие одним и тем же временным меткам, и удалить записи, в которых отсутствуют переменные. С основными характеристиками набора данных можно ознакомиться в таблице 1.

Таблица 1: Параметры набора данных NDBC

Размер данных (сжатых)	512 MB
Размер данных	2851 MB
Количество метеобуев	25
Временной интервал	10 лет (2010–2019)
Общее количество записей	≈ 1,6 млн.



Фигура 2: Производительность программы, написанной адаптированном Guile и C++ на нескольких узлах кластера

Алгоритм следующий. Программа сканирует входную директорию на наличие файлов и группирует их по году и номеру станции. Затем для каждой группы программа считывает файлы и создает хеш-таблицу, в которой ключом является метка времени, а значением - пять массивов с данными для каждого параметра для этой конкретной метки времени. Далее удаляются группы с отсутствующими файлами и записи с отсутствующими массивами. После очистки всех данных программа завершает работу.

Эта программа была написана, используя как низкоуровневый интерфейс C++, так и высокоуровневый интерфейс Guile с ручным параллелизмом, а затем была сравнена производительность двух реализаций. Кроме того, эта программа была реализована на чистом языке Guile, но с использованием параллельных методов, таких как «n-par-map». На рисунке 1

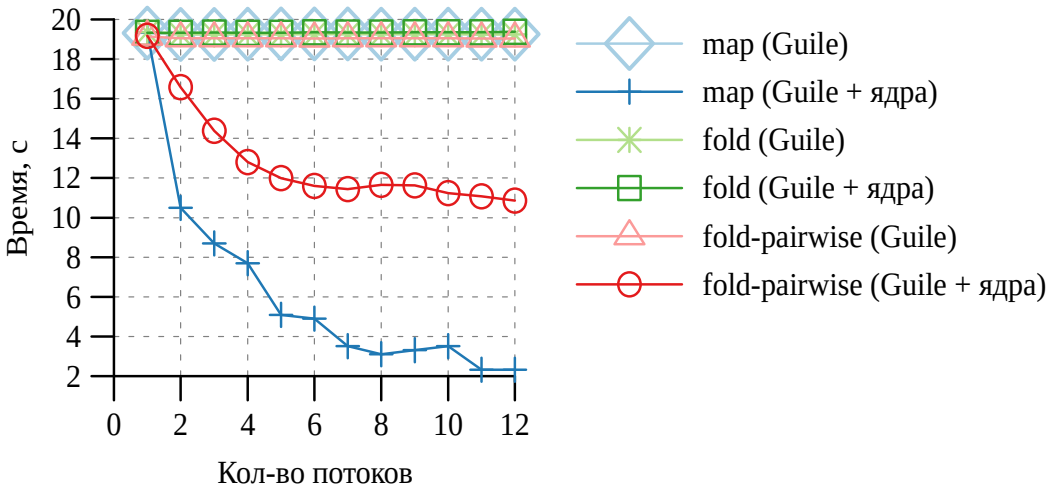
показано, что Guile вызывает заметные накладные расходы по сравнению с C++, но производительность ядер сопоставима со встроенными параллельными формами Guile. К сожалению, накладные расходы, вызываемые параллельным интерпретатором Guile в данной задаче оказались существенно выше всех остальных версий программы, поэтому из этого сравнения такой вариант был исключен. Кроме того, было произведено сравнение работы этих программ, запущенных на кластере. На рисунке 2 показаны накладные расходы при использовании Guile в сравнении с C++. Можно отметить, что накладные расходы уменьшаются при увеличении количества узлов кластера. Параметры кластера приведены в таблице 2.

Таблица 2: Параметры кластера

Процессор	Intel Xeon L5630
Количество узлов	6
Количество процессоров на узле	2
Количество ядер на процессоре	4
Количество потоков на ядро	2
Оперативная память	24 Гб
Сеть	Ethernet 1 Гбит

Для тестирования параллельного интерпретатора Guile были выполнены синтетические тесты производительности, используя формы из листинга 2. Для проверки каждой формы синтетическая процедура, которая «спит» 200 миллисекунд, применялась к списку из 96 элементов. Получившийся скрипт был запущен, используя оригинальный интерпретатор Guile и параллельный интерпретатор, и было измерено общее время работы для разного количества потоков. Для оригинального интерпретатора Guile время работы всех форм одинаково для любого количества потоков. Для параллельного интерпретатора время выполнения «map» и «fold-pairwise» уменьшается с количеством потоков, а время выполнения «fold» остается

неизменным (рисунок 3).



Фигура 3: Зависимость времени работы схем Guile из листинга 2 от количества параллельных потоков и интерпретаторов.

Недостаток автоматического параллелизма - высокие накладные расходы Guile по сравнению с C++. Вероятно, эти расходы слишком велики для написания высокопроизводительных приложений, но приемлемы для написания сценариев и потоков задач. Цель работы состояла в том, чтобы создать однородный программный интерфейс, доступный на нескольких языках, и программист может выбирать, сколько накладных расходов приемлемо для рассматриваемой программы. Поскольку каждый язык использует ядра для параллелизма, это не делает предложенный интерфейс неоднородным (ядра могут быть отправлены между программами, написанными на разных языках), но все же способствует написанию сценариев управления и рабочих процессов на другом языке, чем параллельные программы. Вероятно, компромиссное решение невозможно до тех пор, пока скорость процессоров не сделает накладные расходы незначительными.

7 Выводы

Параллельное вычисление аргументов процедуры - это естественный способ выражения параллелизма на функциональном языке, и в проведенных тестах производительность программы близка к производительности с ручным параллелизмом. Более низкая производительность объясняется тем фактом, что мы вводим больше накладных расходов за счет использования асинхронных ядер для вычисления аргументов процедуры, где это не дает большого прироста производительности (даже при идеальном параллелизме без накладных расходов). Если мы удалим эти накладные расходы, мы получим то же время, что и исходная программа с ручным параллелизмом. Это объясняется тем, что основной цикл программы написан в виде вызова процедуры «map», и наш интерпретатор делает его параллельным. Параллельное выполнение этого цикла дает наибольший прирост производительности по сравнению с другими частями программы. Ожидается, что разница между автоматическим и ручным параллелизмом будет более заметной в более крупных и сложных программах, и в будущем планируется протестировать больше алгоритмов с известными параллельными реализациями.

8 Заключение

Использование аргументов процедуры для определения частей параллельной программы дает новые возможности для написания параллельных программ. В императивных языках программисты переставляют и переписывают циклы, чтобы оптимизировать шаблоны доступа к памяти и помочь компилятору векторизовать код, а в случае с параллельным вычислением аргументов в функциональных языках они могут переписывать формы, чтобы помочь интерпретатору извлечь больше параллелизма. Этот параллелизм является автоматическим и не влияет на результат работы программы. С помощью ядер и кластерного планировщика эти параллельные вычисления превращаются в распределенные вычисления: ядра обеспечивают стандартный способ выражения параллельных и распределенных частей программы, автоматическую отказоустойчивость для главных и рабочих узлов и автоматическую балансировку нагрузки с помощью планировщика кластера. Вместе ядра и параллелизм на основе аргументов обеспечивают низко- и высокоуровневое программирование.

9 Литература

[1] Apache Software Foundation: Hadoop, https://hadoop.apache.org

[2] Apache Software Foundation: Storm, https://storm.apache.org

[3] Brown D. K. et al. JMS: an open source workflow management system and web-based cluster front-end for high performance computing //PLoS One. – 2015. – Т. 10. – №. 8. – С. e0134273.

[4] Dean J., Ghemawat S. MapReduce: simplified data processing on large clusters //Communications of the ACM. – 2008. – Т. 51. – №. 1.

[5] Deelman E. et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems //Scientific Programming. – 2005. – Т. 13. – №. 3. – С. 219-237.

[6] Galassi M. et al. Guile Reference Manual. – 2002.

[7] Gankevich I., Tipikin Y., Korkhov V. Subordination: Providing resilience to simultaneous failure of multiple cluster nodes //2017 International Conference on High Performance Computing & Simulation (HPCS). – IEEE, 2017. – С. 832-838.

[8] Gankevich I. et al. Factory: Non-stop batch jobs without checkpointing //2016 International Conference on High Performance Computing & Simulation (HPCS). – IEEE, 2016. – С. 979-984.

[9] Gankevich I., Tipikin Y., Gaiduchok V. Subordination: Cluster management without distributed consensus //2015 International Conference on High Performance Computing & Simulation (HPCS). – IEEE, 2015. – С. 639-642.

[10] Hutton G. A tutorial on the universality and expressiveness of fold //Journal of Functional Programming. – 1999. – Т. 9. – №. 4. – С. 355-372.

[11] Islam M. et al. Oozie: towards a scalable workflow management system for hadoop //Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies. – 2012. – С. 1-10.

[12] Krajca P., Vychodil V. Software transactional memory for implicitly parallel functional language //Proceedings of the 2010 ACM Symposium on Applied Computing. – 2010. – С. 2123-2130.

[13] Lattner C., Adve V. LLVM: A compilation framework for lifelong program analysis & transformation //International Symposium on Code Generation and Optimization, 2004. CGO 2004. – IEEE, 2004. – С. 75-86.

[14] Lawrance N. R. J. et al. Ocean deployment and testing of a semi-autonomous underwater vehicle //OCEANS 2016 MTS/IEEE Monterey. – IEEE, 2016. – С. 1-6.

[15] Lord T. An Anatomy of Guile: The Interface to Tcl/Tk //Tcl/Tk Workshop. – 1995. – С. 95-114.

[16] McCarthy J. Recursive functions of symbolic expressions and their computation by machine, part I //Communications of the ACM. – 1960. – Т. 3. – №. 4. – С. 184-195.

[17] Pinho E. G., de Carvalho Junior F. H. An object-oriented parallel programming language for distributed-memory parallel computing platforms //Science of Computer Programming. – 2014. – Т. 80. – С. 65-90.

[18] Rabhi F. Patterns and skeletons for parallel and distributed computing. – Springer Science & Business Media, 2003.

[19] Stewart R., Maier P., Trinder P. Transparent fault tolerance for scalable functional computation //Journal of functional programming. – 2016. – Т. 26. – С. e5.

[20] Sussman G. J., Steele G. L. The first report on Scheme revisited //Higher-Order and Symbolic Computation. – 1998. – Т. 11. – №. 4. – С. 399-404.

[21] Vavilapalli V. K. et al. Apache hadoop yarn: Yet another resource negotiator //Proceedings of the 4th annual Symposium on Cloud Computing. – 2013. – С. 1-16.

[22] Wilde M. et al. Swift: A language for distributed parallel scripting //Parallel Computing. – 2011. – Т. 37. – №. 9. – С. 633-652.

[23] Zaharia M. et al. Apache spark: a unified engine for big data processing //Communications of the ACM. – 2016. – Т. 59. – №. 11. – С. 56-65.

[24] Earle, M. D. Nondirectional and directional wave data analysis procedures //NDBC technical Document. – 1996.