И.Г. Захарова, Н.Д. Гибаев
Тюменский государственный университет, г.Тюмень
УДК 532.546.2

Разработка и программная реализация механизма обеспечения отказоустойчивости взаимосвязанных расчетов в распределенной системе

Аннотация. Рассмотрено создание высокоуровневой системы обработки массивных данных на основе философии фреймворка OTP (Open Telecom Platform) языка Erlang. Разработан фреймворк OctopusProject с высокоуровневыми абстракциями, использующий блокирующую очередь для балансировки нагрузки и соблюдения принципа "обратного давления", СУБД mnesia для обеспечения отказоустойчивости. Для поддержания целостности исходной задачи использовано дерево редукций параллельно исполняемых заданий [1].

Ключевые слова: Erlang, OTP, распределенные системы, отказоустойчивость, связность данных, математическое моделирование.

Введение

При решении задач обработки больших потоков данных прибегают к массивному распараллеливанию вычислений. При этом эффективному распараллеливанию поддаются либо атомарные, либо слабосвязные данные, а выполнение вычислительных задач с общими, либо смежными данными не вносит значительного влияния на итоговый результат. В случае же сильносвязных входных данных (в частности, с определенной функциональной зависимостью) возникает проблема поддержания целостности исходной задачи. Это связано с тем, что при распараллеливании вычислительной нагрузки вычисления на отдельном узле не имеют сведений о *контексте* вычислений. Как следствие, результат работы вычислительной системы (программы) становится непредсказуемым, и в данном случае речь уже идет о конкурентных вычислениях, так как возникает проблема синхронизации узлов [2]. При программной реализации таких вычислительных систем требуется поддерживать как параллелизм на уровне узлов, процессов и потоков,

так и целостность исходной задачи путем синхронизации результатов вычислений. Частичное упорядочение вычислительных заданий достигаться может множеством путей [3], но каждая итерация синхронизации является сложным и уязвимым процессом, что превращает этап синхронизации в единую точку отказа. Все это определяет актуальность обеспечения отказоустойчивости системы для гарантированного получения результата.

Таким образом, целью данной работы является создание абстрактной распределенной системы, обеспечивающей целостность и эффективность исполнения исходной задачи, а также достижение отказоустойчивости всей системы в целом. В процессе создания системы, отвечающей сформулированным требованиям, можно определить следующие этапы: выделение и разработка обобщенных необходимых абстрактных элементов для исполнения вычислений и для обеспечения их исполнения; покомпонентное тестирование и оптимизация; разработка механизма для контроля разделяемых между процессорами данных и их сопряжения; тестирование системы на примере решения задачи Дирихле с декомпозицией области решения.

Предлагаемый подход

Сформулированным требованиям удовлетворяет ряд открытых решений на базе MPI, Hadoop и Scala. Однако они имеют недостатки, связанные, в первую очередь, с отсутствием общего эффективного решения проблемы отказоустойчивости [4], сложностью разработки и конфигурирования, необходимостью программной балансировки нагрузки. Очевидны главные преимущества описанных платформ - внушительная кодовая база, использование системных языков при реализации программ позволяет крайне эффективно работать с памятью.

В работе предложено реализовать для решения вышеописанных проблем абстрактный программный каркас OctopusProject на базе языка Erlang и философии стандарта OTP при исполнении процессов внутри виртуальной машины HiPE. Данное решение будет менее эффективным с точки зрения исполнения вычислений [5], но предоставит большой набор готовых абстракций для разработки распределенной конкурентной системы и обеспечения отказоустойчивости. Для разрешения проблемы потери эффективности числовых операций целесообразно вынести исполнение таких операций за пределы виртуальной машины с помощью

NIF - Native Implemented Function [6].

Выделение обобщенных компонент

Анализ опыта [7] позволяет вообще полностью абстрагироваться от конкретной предметной области получаемого решателя, поскольку необходимым требованиям в принципе удовлетворяет гораздо более широкий класс помимо задач математического моделирования. При попытке генерализовать требования для систем массивного распараллеливания вычислительной нагрузки были выделены следующие необходимые функции:

1. Инициализация компонент системы.
2. Обработка ввода.
3. Разделение входного массива данных на части и формирование из этих частей отдельных заданий для процессоров.
4. Назначение каждому процессору задания для расчетов.
5. Получение от каждого процессора результатов вычислений.
6. Анализ каждого результата на минимизацию расхождений в разделяемых данных между несколькими заданиями. Если разница норм результатов в разделяемых данных двух заданий не превосходит некоторой величины, то данные задания считаются *относительно близкими в разделяемых данных* и могут быть объединены в одно задание - результаты вычислений старых заданий переходят в новое задание. В противном случае, данные задания считаются *относительно далекими в разделяемых данных* и к каждому заданию применяется *функция мутации* для повторной отправки на вычисление процессорам.
7. Шаг 6 исполняется до тех пор, пока вычислительные задания не объединятся в одно
8. Вывод данных

Как видно, необходимые функции абстрактны и являются очень обобщенным программным каркасом.

При указанной генерализации основных функций для решения конкретной задачи необходимо реализовать функции интерпретации и разделения входного потока данных, сравнения, слияния и мутации разделяемых данных.

Для облегчения понимания абстрактных и конкретизируемых функций в контексте системы на базе Erlang / OTP нужно ввести понятие *процесса*, под которым понимается функция или последовательность функций, приводящих к изменению состояния системы. *Состояние системы* характеризуется как внутренними данными, так и наличием или отсутствием тех или иных процессов [8]. С точки зрения процесса, реализация программы может рассматриваться уже не как последовательное моделирование параллельно исполняемых действий, а как полностью асинхронное [9], декларативное описание переходов из одно состояние в другое и потоковое движение данных.

Поскольку для распределенных систем семантика процессов подходит органичнее, чем семантика объектов, можно описать декларативно течение процессов в системе и коммуникаций между ними. Такую конкуррентную модель называют моделью акторов [10]. Архитектура, представленная как набор акторов, более наглядна с точки зрения построения компонент и обеспечения их отказоустойчивости, где каждый из них - процесс, где отсутствует общая среда для передачи данных и имеется только межпроцессная коммуникация.

Архитектура компонент

Для определения в семантике процессов абстрактных и конкретных функций, описанных в разделе “Выделение обощенных компонент”, можно выделить компоненты-процессы, представленные на рисунке 1.

Определение. Задача - это абстракция, определяющая набор из необходимых данных для атомарного расчета на одном процессоре. Задача определяется по её *метке*.

Определение. Ведро - набор ключевых элементов.

Input Facility - Обработчик ввода: принимает входной поток данных и интерпретирует в этом потоке отдельные ключевые части. Позволяет продуцировать

записи, которые уже в дальнейшем используются внутри остальных процессов. Эти записи могут генерироваться в ходе, в частности, считывания из файла или обращения по URL, где входной поток определенным образом поддается синтаксическому анализу и из него можно вычленять необходимые части.

Distributed Data Storage - Распределенное хранилище: используется для долгосрочного хранения записей. Все основное состояние среды выполнения хранится только в распределенном хранилище. Поэтому можно интерпретировать этот компонент как способ централизации распределенной системы.

Task Generator - Генератор задач: подготавливает стартовые задачи для вычислений и формирует ведра, после чего связывает задачу с соответствующим ведром. Генератор задач опрашивает по заданной метке записи в распределенном хранилище и проверяет готовность ведер в момент времени. В случае, определенные ведра готовы (соответствующие ему записи загружены в хранилище), то соответсвующее им задание на расчеты становится готовым к исполнению. Когда задание готово, оно отправляется на назначение процессору через *брокера*.

Broker - брокер задач: аккумулирует в себе задания на исполнение и назначает их процессорам. Поскольку в общем случае вычислительных задач много больше чем процессоров в системе, встает вопрос балансировки нагрузки. Для исполнения *получения во владение* задания (task acquiring в семантике OctopusProject) используется как раз этот процесс. Балансировка нагрузки достигается использованием очереди заданий без приоритета.

Worker - исполнитель задач: непосредственно запускает функцию-обработчик над данными в задании. После окончания вычислений процесс-исполнитель записывает слепок результатов в хранилище и отправляет эти результаты на *сопряжение*. Процессы-исполнители являются динамическими, они запускаются только при надобности брокером. В случае возникновения исключительного поведения в ходе обработки задания процесс-исполнитель будет перезапущен.

Conjugator - компонент сопряжения разделяемых данных: осуществляет мутацию или слияние результатов задания. После завершения задания исполнителем, необходимо восстановить контекст целевой задачи, что и делает *Conjugator*. Данный процесс является динамическим и включается только по требованию процесса-исполнителя. Для получения сведений о наличии разделяемых данных Conjugator запрашивает граф связей между ведрами у хранилища.

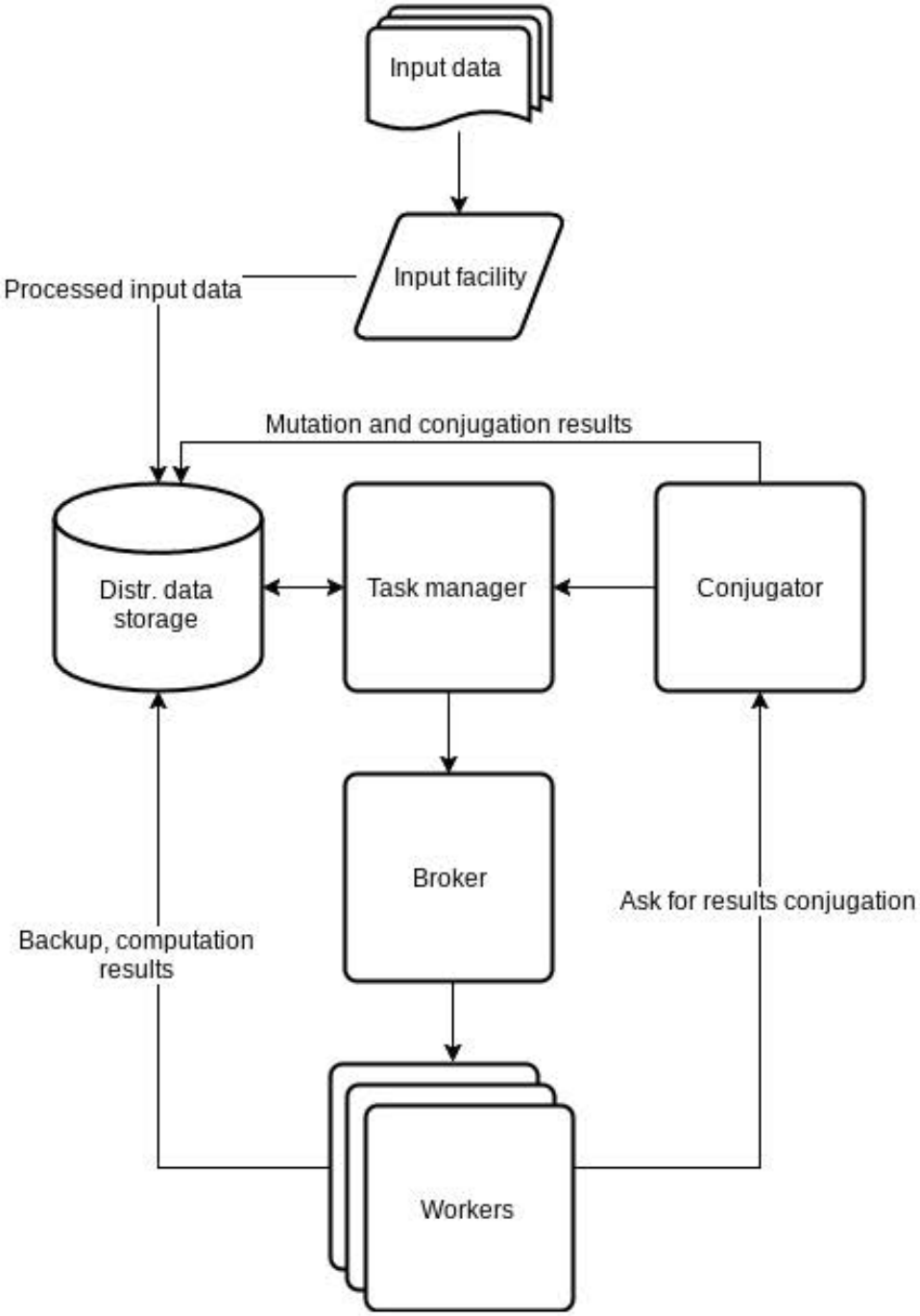

Рис. 1: Архитектура OctopusProject

Средства отказоустойчивости

Обзор отказов в результирующей системе представлен в [7], также там рассматриваются возможные причины возникновения отказов в распределенной системе. Большинство принципов описанных в работе [7] так же применимы и к OctopusProject, но лишь с некоторыми модификациями.

Использование семантики процессов облегчает обеспечение отказоустойчивости, потому что изменяет привычный способ восприятия отказов в системе на этапе проектирования. Поскольку Erlang/OTP разрабатывался в первую очередь как средство для программирования устройств телекома, он реализует большинство принципов и в других программах [8].

Для выполнения требований к отказоустойчивости (устойчивость к отказам в сети, аллокации памяти, отказам конкретного физического узла) можно использовать следующие возможности Erlang/OTP.

Децентрализованная виртуальная машина

Каждый экземпляр виртуальной машины HiPE запускается один раз на каждом физическом узле кластера и переходит в состояние демона. При конфигурировании виртуальной машины перечисляется список имен узлов которые потом могут быть доступны по имени. Виртуальная машина использует по умолчанию все доступные машинные ресурсы на узле и разворачивает планировщик. Исполнение байткода на узле в дальнейшем идет асинхронно [9]. Коммуникационная прозрачность, в свою очередь, предлагает абстрагирование от вида порождаемого процесса. Вне зависимости от того, на каком из узлов в системе работает процесс, коммуникация с ним методом передачи сообщений остается такой же. В широком смысле, все узлы в системе получаются равноправными, поскольку ни один из них не может быть классифицирован как “мастер-узел”. Отсутствие централизации приводит к устойчивости к машинным отказам на узлах и слабой связности. Исполнение байткода вместо низкоуровневых машинных инструкций ведет к устойчивости к отказам в системном смысле, поскольку, в частности, проблемы аллоцирования памяти не приводят к усложнению логики процессов.

Mnesia

OTP может восприниматься уже не как простой программный каркас, а как цельная система и набор инструментария для проектирования отказоустойчивых распределенных приложений. В Erlang в стандартной библиотеке имеется встроенная распределенная NoSQL СУБД реального времени *mnesia*, которая предоставляет возможности к быстрым и поступным репликации и развертыванию. В проекте OctopusProject компонент хранилища построен полностью вокруг функциональных возможностей mnesia, поскольку на ней лежит серьезная ответственность за хранение слепков данных всех остальных компонент. Централизация движения потоков данных в контексте системы OctopusProject не рассматривается как сведение к единой точке отказа, напротив, децентрализованная природа самого устройства mnesia размывает точку отказа, поскольку остановить работу всей СУБД или повредить данные в ходе отсоединения одного из узлов системы не представляется возможным. Mnesia по умолчанию разворачивается на всех физических узлах системы, но точка доступа к ней всего одна. Использование СУБД, встроенной в виртуальную машину уменьшает коммуникационные издержки для доступа к данным, что ведет к теоритическому росту производительности. Использование NoSQL организации хранилища и способов доступа к нему приводит к упрощению схемы представления записей [11].

Supervision tree

В работах [7] и [4] рассматривались подходы к реализации отказоустойчивости, стандарт OTP реализует среди описанных принцип использование процессов-супервизоров. Эти процессы наблюдают за процессами-детьми, чтобы в случае их ненормального завершения собрать необходимые данные и перезапустить процесс для продолжения работы системы. Объединение процессов-наблюдателей в древообразную структуру позволяет разграничивать правила для реагирования на отказы в системе, пример дерева супервизоров представлен на рисунке 2.

Backpressure principle

При длительной высоконагруженной работе системы, вычислительные узлы испытывают значительную машинную нагрузку при обеспечении и исполнении вы-

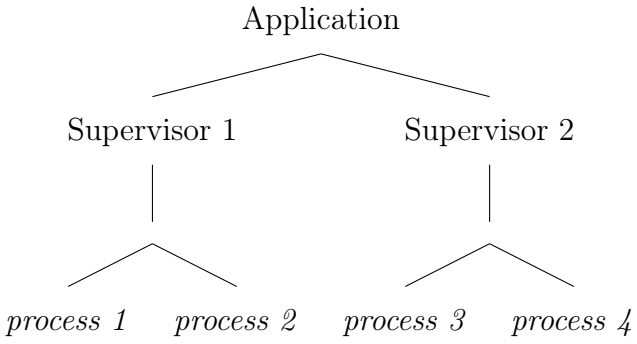

Рис. 2: Пример представления процессов и их наблюдателей в OTP

числений. Поэтому при анализе распределенных систем одним из значимых видов отказа выделяют принцип “обратного давления”. Он гласит, что отказоустойчивая распределенная система при движении потоков данных всегда должна работать с такой же скоростью, какая скорость у самого медленного компонента. Использование этого принципа предполагает защиту распределенной системы от переполнения буферов ввода-вывода и коммуникаций. Одной из распространненых проблем, которая возникает в системах, не реализующих данный принцип, является утечка памяти в очереди планировщика [12].

В OTP реализация базового механизма обратного давления реализуется с помощью блокирующих операций *call*, которые не позволяют текущему процессу продолжать дальнейшее исполнение кода, пока отправитель не получит разрешение на прием от получателя и результат вызова [13].

Safe NIF

Native Implemented Functions, исполняющиеся за пределами HiPE планировщика в виде нативного кода, являются одним из способов значительной оптимизации производительности виртуальной машины, но у этих функций есть серьезный недостаток - при любом исключении в ходе исполнения функции выключится вся виртуальная машина HiPE, что делает использование NIF опасным решением.

В OctopusProject используется обертка Rustler [14], которая позволяет встраивать в виртуальную машину функции, написанные на языке Rust. Rustler в случае возникновения исключений перехватывает их до достижения ядра планировщика и преобразует их в стандартные ошибки внутри HiPE, которые могут быть легко обработаны.

Безопасная передача сообщений

Передача сообщений внутри виртуальной машины HiPE может осуществляться в двух возможных способах - “Fire And Forget”, когда сообщение просто добавляется в *почтовый ящик* процесса, либо же способом мониторинга, когда процесс блокируется до получения уведомления от получателя. Второй способ гораздо безопаснее, поскольку учитывает так же и тот факт, что процесс внутри виртуальной машины может быть перезапущен и внутри него могут быть другие данные.

Балансировка нагрузки

Для балансировки нагрузки между узлами кластера используются инструменты *erl_pool* и блокирующая очередь. *Erl_pool* - это модуль в OTP, который позволяет осуществлять базовую балансировку нагрузку посредством создания пула исполнителей. На выбор кандидата из пула влиют размер его отложенных сообщений и системная нагрузка узла. Блокирующая очередь позволяет обеспечивать простой механизм назначения вычислительных заданий узлам в кластере при условии сохранения возможность горизонтального масштабирования системы. Использование механизма блокировок обеспечивает дополнительную отказоустойчивость, защищая процесс очереди от переполнения памяти.

Поддержание целостности задачи

Механизм обеспечения целостности исходной задачи базируется на применении операции *редукции дерева сопряжений*, сворачивающей подобласти задачи в более крупные на основе слияния соответствующих ведер.

Общие положения

Пусть в общем случае на вход для декомпозиции исходной задачи поступает массив ключей $K = [k_1, k_2, \ldots, k_\mathrm{n}]$ и массив значений $P = [p_1, p_2, \ldots, p_\mathrm{n}]$, где между этими массивами устанавливается отношение $k_\mathrm{i} \rightarrow p_\mathrm{i}$ где $i \in [1, n]$. Для поддержания целостности выдвигается гипотеза о таком разбиении области на перекрывающиеся подобласти порядка o на m ведер, где o - величина перекрытия исходной области задачи. Тогда получаем массив ведер $B = [b_1, b_2, \ldots, b_\mathrm{m}]$,

$b_{\text{i}} = [k_{l_i}, k_{l_{i+1}}, \ldots, k_{r_i}]$, $i \in [1, m]$, тогда размер ведра $size\_b_i = Len(b_i) = r_{\text{i}} - l_{\text{i}}$. Перекрытие между двумя ведрами $o_{\text{i}} = b_{\text{i}} \bigcap b_{\text{i+1}}$ для $i \in [1, m-1], Len(o_{\text{i}}) = o$. Другими словами, любые ведра $b_{\text{i}} = [k_{l_i}, k_{l_{i+1}}, \ldots, k_{r_i}]$ и $b_{\text{j}} = [k_{l_j}, k_{l_{j+1}}, \ldots, k_{r\_j}]$ имеют разделяемые данные, если выполняются условия $i \neq j$, $r_i \geq l_j$.

Соседними ведрами b_{i} по отношению b_{j} будем называть те ведра, где $b_{\text{i}} \bigcap b_{\text{j}} \neq \emptyset$ и $i \neq j$. Очевидно, что разбиение области на m ведер с порядком $o = 0$ представляет собой разбиение B, где каждое ведро имеет структуру $b_{\text{i}} = [k_{l_i}, k_{i_{i+1}}, \ldots, k_{r_i}]$ и для любой пары ведер b_{i} и b_{j} выполняются условия $i \neq j, r_{\text{i}} < l_{\text{j}}$.

Если исходная задача требует поддержания целостности (то есть, в частности, определеная некоторая функциональная зависимость данных, доступных по одним ключам, от других, доступных по другим), то тогда порядок ее разбиения всегда больше нуля. Такую исходную задачу назовем **связной**. Если же исходная задача не требует поддержания целостности (то есть, между никакими данными невозможно и не нужно определять зависимость, например, это массивные данные из лог-файлов), то тогда ее порядок разбиения равен нулю и только нулю. Такую исходную задачу назовем **несвязной** или **просто параллелизуемой**.

В данной работе рассматривается связный класс задач с фиксированным размером всех ведер $size\_b$(либо с фиксированным для всех кроме одного) с порядком разбиения $0 < o <= \lfloor \frac{size\_b}{2} \rfloor$. При таком разбиении у любого ведра b_{i} может быть по два соседних ведра при $i \in [2, m-1]$, а b_1 и b_{m} будут иметь по одному соседу.

В таких задачах количество ведер определяем по формуле 1.

$$\left\lceil \frac{size\_b + (\lfloor \frac{Len(K)}{size\_b - o} \rfloor - 1) * (size\_b - o)}{Len(K)} \right\rceil \tag{1}$$

Относительно близкими ведрами назовем те ведра, для которых некоторым образом определенная функция *ExitMergeReason(b_i, b_j)* возвращает истину, природа этой функции не имеет значения на текущий момент. Если же функция *ExitMergeReason(b_i, b_j)* возвращает ложь, то такие ведра назовем **относительно далекими**.

Слиянием двух ведер назовем некоторую функцию *Merge(b_i, b_j)*, объединяющую ключи из двух ведер в одно. Разделяемые между ведрам ключи отбираются внутренним механизмом, что не принципиально.

Таким образом получаем, что процесс объединения ведер результатов вычислений должен быть рекурсивным процессом объединения исходного разбиения в

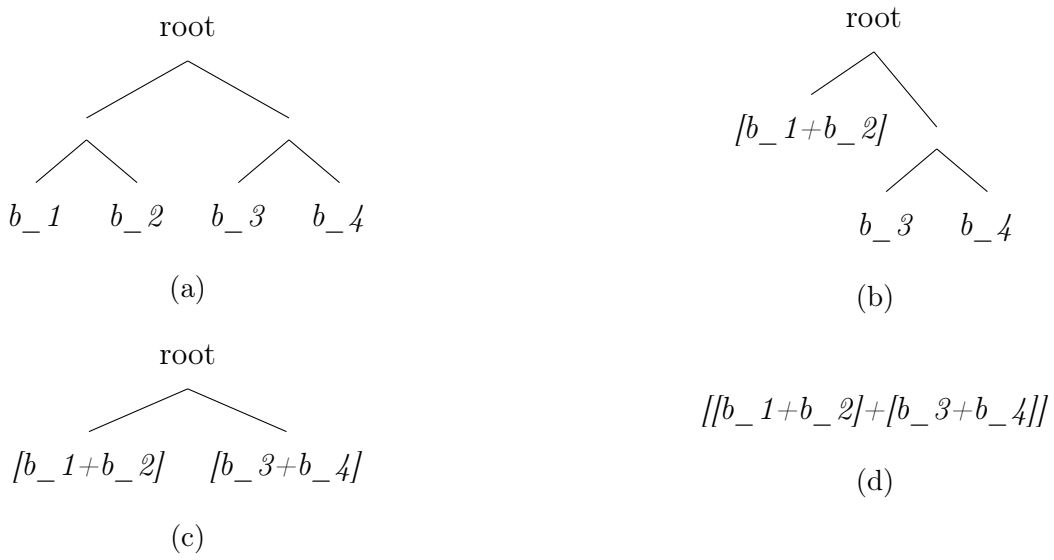

Рис. 3: Пример свертки дерева редукции. (a) - начальное дерево, (b) - свертка ведер b_1 и b_2, получение нового ведра $[b_1{+}b_2]$, (c) - свертка ведер b_3 и b_4, получение нового ведра $[b_3{+}b_4]$, (d) - свертка корневого элемента, окончание сопряжений

более крупные до восстановления всей исходной задачи.

Дерево редукций

Для описания порядка объединения ведер используется древообразная структура данных. В нелистовых узлах не хранится никаких значений, в листовых - ссылки на идентификаторы ведер. Соседние ведра имеют одного общего родителя.

После исполнения слияния двух соседних ведер, ключ нового ведра записывается в их общего родителя, а листья, в которых хранились ссылки на два сопрягаемых ведра, отбрасываются.

Пример итерационного процесса сопряжения ведер представлен на рисунке 3.

Алгоритм сопряжения

После завершения процессом-исполнителем вычислительного процесса, *Worker* вызывает у *Conjugator* функцию сопряжения *Conjugate(b_i, b_j, ConjTree)*, принимающую в качестве аргументов ведра b_i и b_j и дерево сопряжений *ConjTree*. Поскольку процессная логика не имеет понятия состояния, функция свертки листьев дерева продуцирует новое дерево со сброшенными старыми листьями. Формально алгоритм описан на листинге 1 в соответствии с нотацией псевдокода IEEE.

После исполнения функции *Conjugate* процесс *Conjugator* сопоставляет полу-

Листинг 1 Алгоритм сопряжения двух ведер

```
1: function CONJUGATE(b_1, b_2, ConjTree)
2:     some initialization goes here
3:     if ExitMergeReason(b_1, b_2) then
4:         NewB ← Merge(b_1, b_2)
5:         NewConjTree ← ReduceTree(NewB, b_1, b_2)
6:         return [conjugated, NewB, NewConjTree]
7:     else
8:         MutBuckets ← Mutate(b_1, b_2)
9:         return [not_conjugated, mut_buckets]
10:    end if
11: end function
```

ченный результат с шаблоном. Если сопряжение прошло успешно, то нужно собрать новое задание на сопряжение уже следующей пары объединенных ведер. Это хорошо видно на рисунке 3: после исполнения (3.b) сопряжены b_1 и b_2, но новое задание на сопряжение объединенных ведер $\lceil b_1{+}b_2 \rfloor$ и $\lceil b_3{+}b_4 \rfloor$ производить нельзя - ведра b_3 и b_4 еще не сопряжены. Поэтому новое задание на сопряжение не генерируется и не отправляется брокеру. После исполнения (3.c) сопряжены уже b_3 и b_4, в результате получается новое ведро $\lceil b_3{+}b_4 \rfloor$, причем их сестренский узел является так же листом (результатом сопряжения) - значит, на уже этом шаге можно собрать новое задание на сопряжение, но уже у корня, как это видно на (3.d).

То есть, обобщая все вышесказанное, можно сделать следующий вывод: при каждом успешном сопряжении относительно близких ведер после редукции дерева сопряжений необходимо производить проверку сестринского узла - если он является листом, то необходимо сгенерировать новое задание на сопряжение.

Описание реализации абстрактных компонент системы на примере численного решения задачи Дирихле для уравнения Пуассона

Для тестирования абстракного решателя была выбрана задача численного решения первой краевой задачи стационарного уравнения теплопроводности. Для относительно “хороших” областей она хорошо поддается декомпозиции и эффек-

тивному сопряжению результатов решения. В данной работе не рассматривается физической смысл используемого алгоритма сопряжения конкретно для уравнения теплопроводности, поскольку для OctopusProject превалирует важность обеспечения этих расчетов.

Декомпозиция области

Как описано в разделе “Поддержание целостности задачи”, декомпозиция исходной задачи производится на области, но расположение этих областей зависит от *связности* задачи. Численное решение уравнения теплопроводности может производиться с помощью аппроксимации дифференциального уравнения разностным, в частности, с помощью схемы Ричардсона “крестом” [15]. Наличие схемы определяет четкую функциональную зависимость соседних ведер из-за функциональной зависимости между значениями в разделяемых данных.

Формализуем исходной задачи в семантике *OctopusProject*. Для этого сгенерируем сетку в виде двумерного массива $M = [m_1, m_2, \ldots, m_n]$. Каждый элемент m_i представляет собой строку двумерного массива. Как было указано в разделе “Поддержание целостности задачи”, элементы m_i переходят элементы v_i массива значений V, а массив K будет состоять из элементов $k_i = i$, соответствие между ними устанавливается $k_i \to v_i$. Другими словами, исходный двумерный массив сохраняется внутри системы *построчно*. Исходная задача, как было указано в разделе “Поддержание целостности задачи”, является связной, поэтому разбиение области будет иметь порядок o больше нуля. Пример такого разбиения “лентами” представлен на рис. 4.

Организация сопряжений

После декомпозиции производится построение дерева сопряжений.

Функция, проверяющая относительную близость ведер представлена на листинге 2.

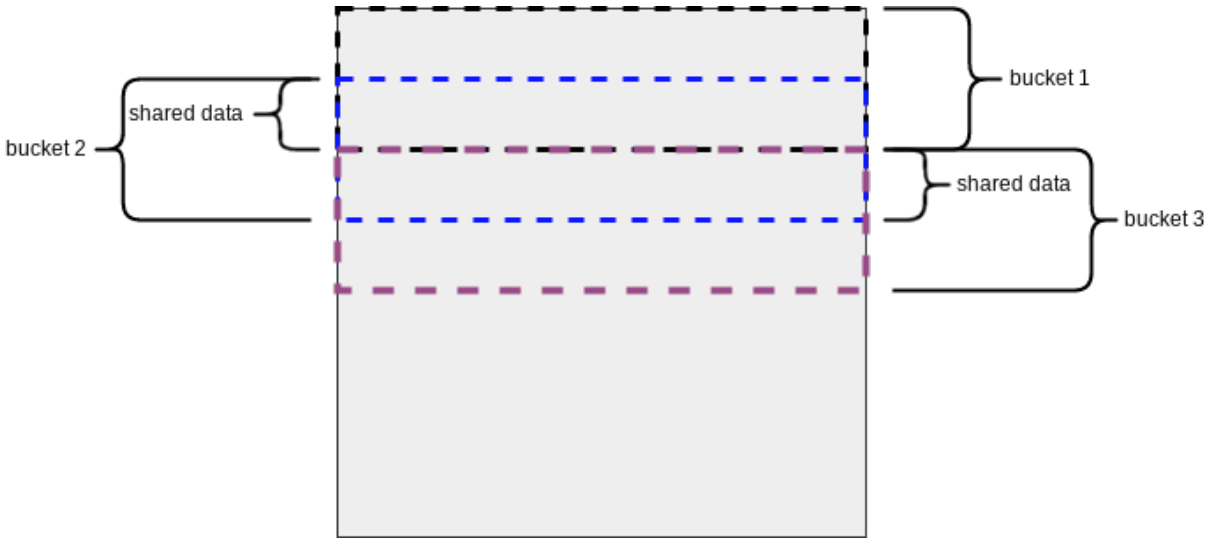

Рис. 4: Пример разбиения области решения исходной задачи Дирихле на “ленты”. Как видно на рисунке, у каждой подобласти (кроме крайних) есть по два соседа.

Листинг 2 Алгоритм проверки на относительную близость ведер

```
1: function ExitMergeReason(b_1, b_2, shared_keys)
2:     v1 ← UnpackBucket(b_1)
3:     v2 ← UnpackBucket(b_2)
4:     v1 and v2 filtered from nonshared keys by mutation this lists
5:     for key ∈ SharedKeys do
6:         MaxDiff ← difference between corresponding elements by Abs(v1[k][i] − v2[k][i])
7:     end for
8:     return MaxDiff < eps
9: end function
```

Функцию слияния ведер определим по правилу “черепицы” - среди разделяемых ключей выберется тот ключ из того ведра, у которого порядковый номер больше (см. листинг 3).

Листинг 3 Алгоритм слияния ведер в одно

```
1: function MERGE(b_1, b_2, shared_keys)
2:     new bucket initialization in NewBucket
3:     k1 ← UnpackBucketKeys(b_1)                          ▷ extract bucket keys
4:     k2 ← UnpackBucketKeys(b_2)
5:     k1_filtered ← ListSubstract(k1, shared_keys)   ▷ remove keys with shared data from first
   bucket's keys list
6:     for key ∈ k1_filtered do
7:         NewBucket[key] ← k1[key]
8:     end for
9:     for key ∈ k2 do
10:        NewBucket[key] ← k2[key]
11:    end for
12:    return NewBucket
13: end function
```

Функцию мутации данных в ведрах после неудачного сопряжения определим достаточно условно - как вычисление среднего арифметического между значениями соответствующих узлов сетки из каждого ведра. Алгоритм представлен на листинге 4.

Листинг 4 Алгоритм мутации двух ведер

```
1: function MUTATE(b_1, b_2, shared_keys)
2:     initialize new bucket tuple BucketPair
3:     for key ∈ shared_keys do
4:         initialize empty tuple new_line
5:         for i = 1toN do
6:             NewLine[i] ← (b_1[key][i] + b_2[key][i])/2
7:         end for
8:         b_1[key] ← NewLine                    ▷ replace old lines with new, mutated
9:         b_2[key] ← NewLine
10:    end for
11:    SetBucketToBucketTuple(BucketPair, 1, b_1)
12:    SetBucketToBucketTuple(BucketPair, 2, b_2)
13:    return BucketPair
14: end function
```

Расчеты внутри области

Внутри каждой из подобластей, объединенных в отдельные ведра, пересчет производится явной разностной схемой, алгоритм пересчета не модифицируется. Функция для пересчета внутри области может быть (и рекомендуется) реализована на системном языке для оптимизации операций над числами с плавающей точкой.

Заключение

Система OctopusProject обеспечивает механизмы для эффективного численного решения задач, поддающихся декомпозиции по областям решения или моделям при наличии явных правил сопряжения. На примере показано, что для решения стационарного уравнения теплопроводности в системе OctopusProject достаточно указать функции для сопряжения соседних областей при декомпозиции и функцию для итерационного процесса внутри подобласти. Проблемы балансировки вычислительной нагрузки, развертывания на узлы в кластере, сопряжений соседних областей и поддержания отказоустойчивости разрешаются изнутри, тем самым повышая уровень оперируемых абстракций.

ЛИТЕРАТУРА

[1] Непомнящих С.В. Методы декомпозиции области и фиктивного пространства // Новосибирский государственный университет. ИВМ СО РАН. 2008. 29 с.

[2] Lamport L. Time, Clocks, and the Ordering of Events in Distributed system // Communications of the ACM T. V. 21 № 7. 1978. P. 558-565.

[3] Tanenbaum A.S. M. van Steen, Distributed systems, Principles and Paradigms. Pearson Prentice Hall. 2006. P. 687.

[4] Gropp W. Lusk E., Fault Tolerance in Message Passing Interface Programs // The International Journal of High Performance Computing Applications V. 18 № 3. 2004. P. 363-372.

[5] Scalas A. Casu G. Pili P., High-performance technical computing with Erlang // Proceedings of the 7th ACM SIGPLAN workshop on ERLANG. 2008. P. 49-60.

[6] *Erlang NIF module documentation.* `http://erlang.org/doc/man/erl_nif.html`, дата обращения: 12.11.2016

[7] Самборецкий С.С. Гибаев Н.Д. Кашинцев М.О., Решение вопросов балансировки нагрузки и обеспечения отказоустойчивости в распределенной системе сопряжения секторных моделей // Суперкомпьютерные дни в России, Труды международной конференции. 2016. С. 967-977.

[8] Armstrong J. Virding R. Wikstroem C. Williams M., Concurrent programming in Erlang. Prentice Hall PTR. 1996. Second Edition. P. 358.

[9] Johansson E. Pettersson M. Sagonas K., A High Performance Erlang System // Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming. 2000. P. 32-43.

[10] Hewitt C. Bishop P. Steiger R., A Universal Modular Actor Formalism for Artificial Intelligence // Proceedings of the 3rd international joint conference on Artificial intelligence. 1973. P. 235-245.

[11] Mattson H. Nilsson H. Wikström C., Mnesia — A Distributed Robust DBMS for Telecommunications Applications // Proceedings of the First International Workshop on Practical Aspects of Declarative Languages. 1999. P. 152-163.

[12] Huang L. Zhang S. Chen M. Liu X., When Backpressure Meets Predictive Scheduling // IEEE/ACM Transactions on Networking (TON). V. 24, I. 4. 2016. P. 2237-2250.

[13] *Erlang/OTP complete reference.* `http://erlang.org/doc/`, дата обращения: 21.10.2016

[14] *Rustler repo.* `https://github.com/hansihe/rustler`, дата обращения: 25.11.2016

[15] Самарский А.А., Введение в теорию разностных схем. М.: Наука. 1971. 553 с.