УДК 004.4’2, 004.7

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ПОДДЕРЖКИ КЛАСТЕРНЫХ ВЫЧИСЛЕНИЙ ПРИ ИСПОЛЬЗОВАНИИ ТЕХНОЛОГИИ ТРАНСПАРЕНТНЫХ ВЫЧИСЛЕНИЙ

В.Д. Павленко, В.В. Бурдейный
Одесский национальный политехнический университет
Украина, 65044, Одесса, просп. Шевченко, д. 3
E-mail: pavlenko_vitalij@mail.ru, vburdejny@gmail.com

Ключевые слова: кластерные вычисления, инструментальные средства параллельных вычислений, транспарентное распараллеливание

Рассматривается технология транспарентного распараллеливания, основанного на заказах. Показаны основные вопросы, возникающие при реализации данной технологии в виде прикладного инструментария параллельных вычислений. Реализация такого инструментария описана с точки зрения системного программиста и с точки зрения пользователя. Практическое его использование проиллюстрировано примером вычислительно сложной задачи, для которой рассмотрен весь процесс разработки параллельной реализации и запуска.

CLUSTER COMPUTING SOFTWARE TOOLS FOR TECHNOLOGY OF ORDERS BASED TRANSPARENT PARALLELIZING / V.D. Pavlenko, V.V. Burdeinyi (Odessa National Polytechnic University, 1 Shevchenko, Odessa 65044, Ukraine).

Technology of orders based transparent parallelizing is considered. Main questions that arise during implementing this technology as a parallel computing framework are described. An existing implementation is described from the point of view of system developer and of user. Its practical use is illustrated by solving a sample problem, including development its parallel implementation and running it.

1. Введение

Параллельные вычисления активно развиваются в последнее время [1, 2, 3, 4], что обусловлено в первую очередь появлением все более сложных задач, решение которых на современных последовательных ЭВМ за приемлемое время получить невозможно [5, 6]. В частности, одной из важных задач в области параллельных вычислений является задача создания средств разработки параллельных приложений, позволяющих быстро создавать эффективные параллельные программы и распараллеливать существующие последовательные

программы, а также предоставлять пользователю средства поддержки разработки – например, анализа трасс выполнения, отладки параллельных приложений или нахождения узких мест. Для решения этой задачи используются два основных подхода. Первый заключается в предоставлении прикладным программистам средств ручного распараллеливания – как правило, низкоуровневых средств организации потоков, синхронизации или коммуникации между вычислительными узлами. Преимущества такого метода – в возможности создания средств, основанных на широко используемых императивных языках программирования (и потому возможности использования существующих реализаций алгоритмов) и широким возможностям для ручной оптимизации, а недостатки – в сложности тестирования, отладки и поддержки параллельных приложений. Другой подход заключается в создании полуавтоматических и автоматических средств распараллеливания. К таким средствам, в чатсности, относится описанная ранее в ряде работ технология транспарентного распараллеливания [7, 8], основная идея которой заключается в выделении наборов алгоритмов, которые могут на высоком уровне быть описаны одной структурой, и реализации шаблонов параллельных алгоритмов в терминах таких структуры, после чего разработанные шаблоны могут быть использованы для распараллеливания конкретных прикладных алгоритмов. Преимуществом такого подхода является возможность реализации достаточно сложных средств поддержки разработки и выполнения параллельных программ в рамках зафиксированных структур, которая достигается путем сужения класса эффективно распараллеливаемых алгоритмов. Впрочем, при появлении алгоритма, не распараллеливаемого эффективно в рамках ни одной из уже реализованных структур, для него может быть реализована новая структура.

Целью данной работы является реализация технологии транспарентного распараллеливания в виде прикладного инструментария параллельных вычислений. Рассматриваются вопросы, связанные в реализацией и практическим использованием подобного инструментария. Описываются основные архитектурные решения, принятые при реализации, структура созданного инструментария, и рассматривается его использование на практическом примере.

2. Принципы организации кластерных вычислений на основе транспарентного распараллеливания

На данный момент в рамках технологии транспарентного распараллеливания была рассмотрена и реализована одна структура алгоритма, основанная на предположении, что в распараллеливаемой программе выделен набор процедур, в процессе работы которых не модифицируются никакие переменные, кроме параметров и данных, недоступных вне данного вызова процедуры, а входные и выходные параметры передаются по значению. Выполнение программы должно сводиться к выполнению одной из этих процедур.

Первый принцип, на котором основан предлагаемый метод распараллеливания, вводит понятие заказа на вычисления. Заказ на вычисления вводится как единица выполняемой на одном из компьютеров кластера работы,

т.е. объем работы, который обязан быть выполнен на одном из компьютеров кластера полностью и не может быть разбит на более мелкие части. В качестве такого объема работы (заказа) вводится выполнение одной процедуры без выполнения тех процедур, которые она вызывает, а выполнение каждой из них выделяется в отдельный заказ. Для того чтобы определить точку старта программы, одна из процедур должна быть описана как главная. Через параметры этой процедуры должны передаваться исходные данные и результаты вычислений.

Для иллюстрации принципов, положенных в основу данной технологии, рассмотрим схематическое изображение процесса выполнения некоторой процедуры (рис. 1):

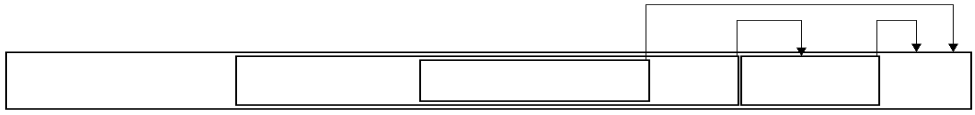

Рис. 1. Схематическое изображение процесса выполнения последовательной процедуры

Здесь прямоугольником будем обозначать время выполнения процедуры, отсчитываемое слева направо. Для обозначения вложенных процедур используются вложенные прямоугольники. Длины прямоугольников и их областей пропорциональны времени, затрачиваемому на выполнение соответствующих частей программы. Линиями изображены связи, которые объединяют моменты вычисления некоторых данных с моментами, когда они впервые используются в дальнейших вычислениях. При этом изображены только связи для данных, вычисляемых во вложенных процедурах, а используемых в основной процедуре или другой вложенной процедуре.

Использование первого принципа для рассматриваемого примера иллюстрируется на рис. 2 (в этом примере считается, что возникающие за время работы четыре заказа будут выполняться на четырех различных процессорах, а их выполнение будет начинаться сразу же). Для наглядности разделенные части одной процедуры также связаны стрелками. Пунктирными линиями очерчены вычисления, производимые на одном процессоре.

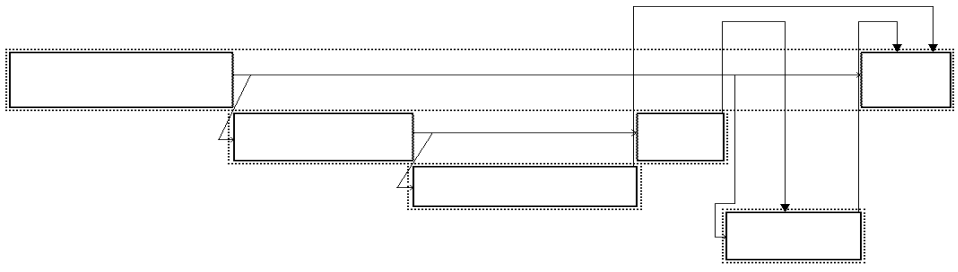

Рис. 2. Схематическое изображение процесса выполнения процедуры при использовании первого принципа

Как видно, хотя в процессе вычислений задействованы четыре процессора, это не позволяет добиться ускорения вычислений, поскольку в каждый момент из них задействован только один. Ускорение же достигается благодаря второму принципу, основанному на существовании отрезков времени между получением некоторых данных и первым их использованием в дальнейших вычислениях,

которые часто могут быть специально увеличены путем изменения порядка вычислений. Такие отрезки времени существуют в большинстве алгоритмов, и именно их наличие предопределяет возможность распараллеливания алгоритма. При традиционном подходе при вызовах процедур вызывающая процедура продолжает свою работу только после окончания работы вызываемой процедуры. Другими словами, вызывающая процедура начинает ожидание результатов выполнения вызываемой процедуры в момент вызова, а заканчивает в момент завершения вызываемой процедуры. Вместо этого предлагается начинать ожидание в момент, когда данные, вычисляемые вызываемой процедурой, потребуются для дальнейшей работы (если в этот момент они уже вычислены, то начинать ожидание не нужно), и заканчивать ожидание в момент, когда эти данные получены. Схематическое изображение двух принципов, положенных в основу данной технологии распараллеливания, приведено на рис. 3.

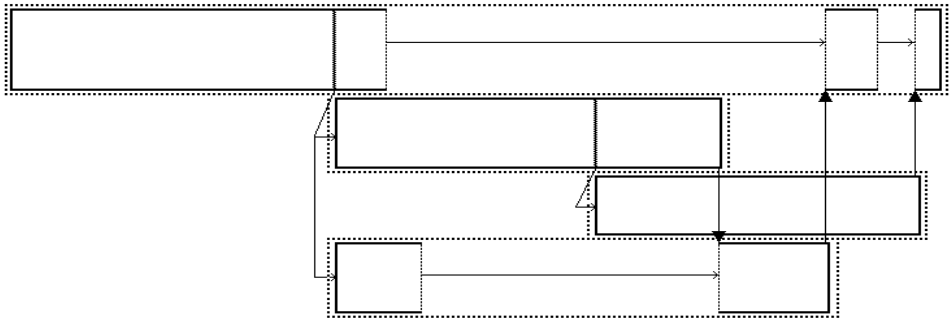

Рис. 3. Схематическое изображение процесса выполнения процедуры при использовании первого и второго принципов

Такой график выполнения может быть получен из предыдущего максимально возможным сдвигом влево всех вычислений, при котором соблюдается условие, что всякое значение используется уже после того, как оно вычислено.

Таким образом, набор выделенных процедур, с одной стороны, используется для введения заказа как единицы работы, а с другой стороны каждая из этих процедур является некоторой частью исходного кода программы, оформленной в силу тех или иных соображений прикладного программиста как процедура в терминах языка программирования. Такое двоякое использование процедур возможно в силу следующих соображений. Если нужно вынести в единицу работы лишь часть процедуры, то ее можно оформить в виде отдельной процедуры. Если не учитывать расходы времени на передачу данных по сети и время работы кода инструментария, реализующего предлагаемую технологию, то слишком мелкое «дробление» кода меняет лишь порядок выполнения процедур, не увеличивая общего времени выполнения программы. Однако с практической точки зрения вынесение в качестве единицы работы часто вызываемых процедур с малым временем работы значительно увеличивает накладные расходы (связанные с передачей данных и работой кода инструментария), поэтому возможность традиционного вызова выделенных процедур должна быть сохранена, и должна использоваться в случаях, когда замена вызова выделенной процедуры на отправку заказа заведомо неэффективна по временным параметрам.

Предлагаемый подход предполагает использование параллелизма заданий и модели MIMD (Multiple Instruction Multiple Data) [9]. Взаимодействие между компьютерами при этом сводится к следующим пяти операциям: получению заказа, отправке заказа, запросам о наличии значений данных, получению вычисленных в другом заказе данных и отправке результатов выполнения заказа. Прикладному программисту из этих пяти операций доступны только две: отправка заказа и получение вычисленных в другом заказе данных. Важно, что детали передачи данных между компьютерами являются несущественными для программиста, а потому могут быть инкапсулированы в инструментарии, реализующем предлагаемую технологию.

В рамках данной технологии выполняемая программа является набором инструкций, описывающих работу кластера в целом, в то время как традиционно использующаяся для написания приложений для кластеров технология MPI [10] основана на том, что программа является набором инструкций для каждого из компьютеров кластера в отдельности. Такой подход, когда программа создается для кластера так, как если бы она полностью выполнялась на однопроцессорном компьютере, во многих случаях предпочтительнее для разработчика прикладных программ. Областью применения предложенной технологии распараллеливания являются вычислительные алгоритмы, реализуемые с использованием параллелизма заданий.

Пример. Для демонстрации предложенной технологии рассмотрим процесс решения задачи исследования информативности формируемых диагностических признаков с помощью процедуры полного перебора [6]. Условно алгоритм решения данной задачи может быть представлен псевдокодом на рис. 4, а.

```
function best_combination(ψ) {
  φ←func1(ψ)
  q←func2(ψ, φ, i)
  for (i=2 ... N)
    q←q∘func2(ψ, φ, i)
  return q
}
```

(а)

```
function best_combination(ψ) {
  φ←func1(ψ)
  for (i=1 ... N)
    α[i]←func2(ψ, φ, i)
  return α[1]∘α[1]∘...∘α[N]
}
```

(б)

Рис. 4. Варианты псевдокода алгоритма решения задачи

В данном алгоритме сначала функция func1 выполняет первоначальную обработку данных (вычисление математических ожиданий признаков и построение ковариационных матриц), а затем производится N вызовов функции func2, из возвращаемых значений которых определяется искомый набор путем последовательного применения бинарной операции ∘.

Как видно, данный алгоритм не содержит существенных отрезков времени между получением данных и первым их использованием, которые требуются для эффективного распараллеливания. Изменим его так, как показано на рис. 4, б. В этом случае операции внутри цикла могут выполняться параллельно и, соответственно, их следует вынести в отдельную процедуру. Поскольку значение N в данном алгоритме является достаточно большим, вынесем в процедуру не выполнение одной итерации цикла, а выполнение набора итераций вместе с вычислением соответствующей части результата (вид псевдокода при этом останется тем же). Процесс выполнения программы

представлен на рис. 5 (считается, что компьютеры кластера получают заказы с выделенного сервера).

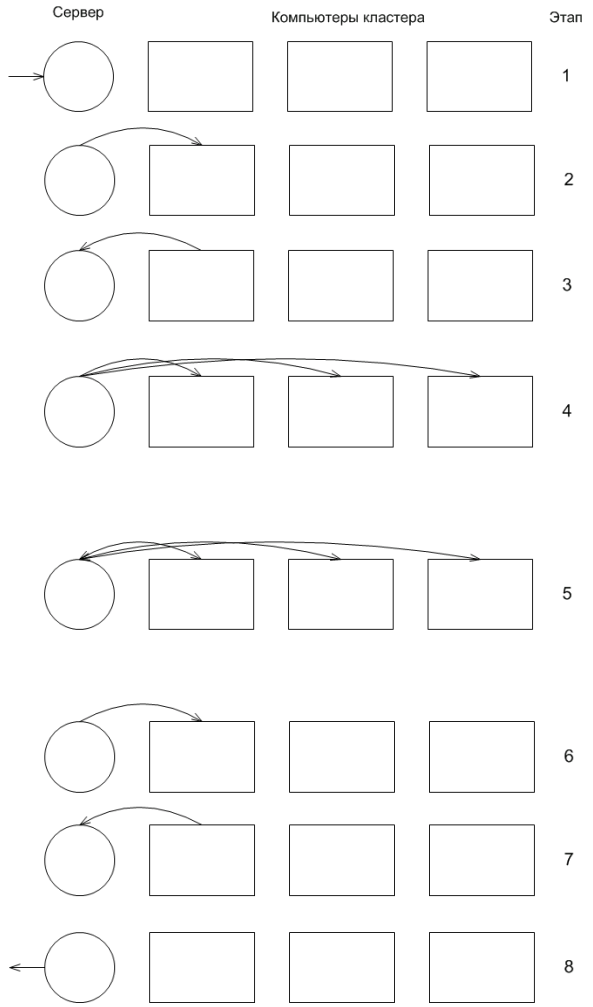

Рис. 5. Взаимодействие компьютеров кластера в процессе решения задачи оценки информативности наборов диагностических признаков методом полного перебора

Предложенный подход к распараллеливанию вычислений может быть реализован на достаточно большом количестве языков программирования. В данной работе используется язык программирования Java, поэтому дальнейшие рассуждения будут производиться в терминах этого языка программирования.

Так как для распараллеливаемой программы было введено требование о том, что в программе выделены процедуры, параметры которых передаются по значению, а в процессе работы какие-либо изменения вносятся только в параметры и временно создаваемые и недоступные извне структуры данных, это позволяет сделать следующие выводы.

Во-первых, передача данных из одной процедуры в другую возможна только через параметры, причем, поскольку параметры передаются по значению, взаимодействие происходит только в два момента времени – в момент вызова и в момент завершения работы вызываемой процедуры. Этим гарантируется, что процедуры во время выполнения не будут работать с какими-либо данными, доступными извне. Таким образом обеспечивается, с

одной стороны, возможность переноса выполнения отдельной процедуры на другой компьютер кластера, а с другой стороны, возможность одновременного выполнения нескольких процедур на одном компьютере, что позволяет после приостановки выполнения какой-либо процедуры использовать освободившиеся вычислительные ресурсы.

Во-вторых, выполнение всякой процедуры сводится к внесению некоторых изменений в переданные значения параметров, а поэтому всякий вызов процедуры можно заменить внесением в переданные значения параметров тех изменений, которые бы выполнила эта процедура в процессе своей работы. Эти изменения могут быть внесены или в момент вызова, или после него, но не позднее первого обращения к значению этого параметра.

Этими соображениями обеспечивается корректность предложенной технологии, понимаемая как совпадение результатов выполнения алгоритмов при отсутствии и при наличии распараллеливания, а также как возможность увеличения скорости выполнения за счет распараллеливанию вычислений.

В терминах языка Java введенные предположения о пользовательском коде означают, что в программе должен быть выделен набор статических методов. Передача параметров по значению вызывает трудности во многих языках программирования. В частности, в Java по значению передаются лишь входные параметры примитивных типов, а организовать передачу по значению можно только для одного выходного параметра примитивного типа, передавая его через возвращаемое значение метода. Поэтому заменим это требование другим, позволяющим сделать те же выводы относительно корректности технологии. Потребуем, чтобы результат и время выполнения процедуры не изменились, если значение каждого из входных параметров не примитивного типа было заменено результатом десериализации сериализованного значения данного параметра, а все значения выходных параметров были заменены значениями по умолчанию.

Средства для отправки заказов могут быть организованы путем генерации набора методов, по сигнатурам совпадающих с выделенными методами, и отправляющих соответствующие заказы на сервер. Методы при этом могут выделяться в исходном коде программы либо при помощи специальных комментариев, либо аннотирования выделенных методов, либо аннотирования методов абстрактного класса именами выделенных методов (заказы на выполнение которых будет отправлять генерируемый потомок этого класса), либо декларирования методов во внешнем файле. Классы с методами для отправки заказов могут быть сгенерированы либо в виде исходного кода, либо непосредственно в виде байт-кода.

2. Программная реализация технологии

Описанная выше технология была реализована в виде прикладного инструментария параллельных вычислений. Его архитектура приведена на рис. 6.

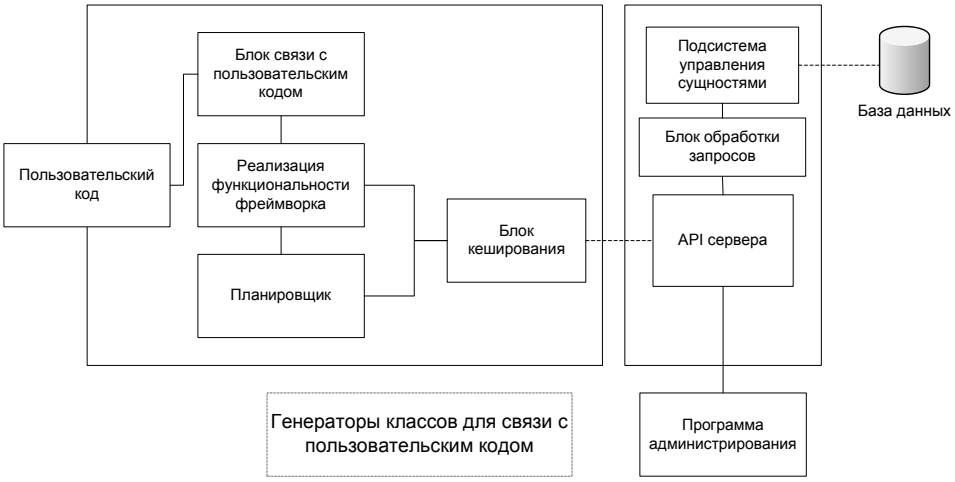

Рис. 6. Архитектура разработанного инструментария

При реализации инструментария была выделена серверная часть, на которую были возложены функции как по координации действий клиентов, так и по хранению промежуточных данных и предоставлению их клиентам. Такое решение было принято для упрощения обеспечения надежности, поддержки динамического изменения состава кластера и управления кластером с единого узла. Дополнительным средством обеспечения надежности является внешняя база данных, которая может использоваться для хранения промежуточных данных вычислений. Это позволяет продолжать вычисления с относительно небольшими потерями времени после аппаратного или программного сбоя сервера или одного из клиентов, поскольку в случае сбоя сервера он может быть перезапущен с восстановлением состояния вычислений из базы данных, а в случае сбоя клиента его задачи могут быть переданы на выполнение другим клиентам.

Серверная часть должна обеспечивать выполнение по запросам от клиентов следующих функций: отправки заказа, получения заказа, отправки значения, получения значения и отправки уведомления о завершении выполнения заказа. Также для отладки параллельных приложений к набору этих функций добавлены функции для получения текущего времени на сервере и вывода отладочных сообщений в создаваемый на сервере файл. Базовыми понятиями, на основе которых организован инструментарий, являются заказ, идентификатор значения и идентификатор неизвестного значения.

Идентификатор значения используется в качестве ссылки на передаваемое значение параметра заказа. Всякое описание заказа содержит лишь идентификаторы, а не сами значения параметров. Это позволяет избегать повторной передачи одних и тех же значений параметров, если они используются в различных вызовах, и тем самым снижать нагрузку на сеть засчет кеширования значений, привязанных к идентификаторам.

Понятие идентификатора неизвестного значения возникает из формулировки второго принципа предложенного подхода: поскольку после отправки заказа выполнение вызвавшей его процедуры должно продолжаться, всякому входному и транзитному параметру этого заказа необходимо присвоить некоторый идентификатор, который затем позволит получить реальное значение соответствующего параметра. Таким образом, результатом выполнения заказа является установка связи идентификатора неизвестного значения каждого из

выходных или транзитных параметров этого заказа с некоторым идентификатором значения или идентификатором неизвестного значения. Возможность установки связи идентификатора неизвестного значения не с идентификатором значения, а с другим идентификатором неизвестного значения не вполне очевидна, но она возникает, например, в ситуации, когда некий заказ в качестве выходного параметра передает значение величины, вычисляемой в другом заказе, и не обращается к этой величине в процессе выполнения.

Понятие заказа возникает как формальное описание информации, требуемой для выполнения заказа. Эта информация включает в себя указание о том, какая процедура должна быть выполнена, информацию о параметрах в момент старта процедуры и параметрах в момент ее завершения. Информация о параметрах в момент старта представляет собой набор идентификаторов значений или идентификаторов неготовых значений, присвоенных входным и транзитным параметрам заказа, значения которых были переданы в момент вызова, замененного отправкой заказа. Информация о выходных параметрах – набор идентификаторов неготовых значений, которые были присвоены выходным и транзитным параметрам заказа. Следует отметить, что подобное представление позволяет достаточно быстро ответить на вопрос о том, совпадает ли информация о входных параметрах двух заказов на выполнение одной и той же процедуры. При условии, что различные идентификаторы значений не могут быть связаны с одним и тем же значением, для этого достаточно потребовать равенства всех соответствующих параметров, причем равенство параметров вводится следующим образом: идентификаторы значений равны, если они совпадают, идентификаторы неизвестных значений или идентификатор значения и идентификатор неизвестного значения – если они связаны. Такое сравнение позволяет в случае детерминированных алгоритмов быстро находить ситуацию, в которой некоторый заказ отправлен в процессе вычислений более одного раза, и вместо последующих выполнений подставлять результаты первого выполнения. В частности, такая ситуация может возникать при повторном решении задачи после внесения изменений во входные данные, если часть вычислений при этом не изменяется.

Кроме клиентской и серверной частей в состав инструментария входит также программа администрирования, позволяющая получать информацию о состоянии кластера и выполнении программ, запускать выполнение параллельных приложений и получать результаты счета, а также программа для генерации служебного кода, используемая при разработке.

Рассмотрим детально подсистемы инструментария, представленные на рис. 6.

Блок взаимодействия с пользовательским кодом и блок генерации исходного кода. Функциональность, доступная пользователю, заключена в нескольких классах. Во-первых, это генерируемый класс библиотеки, который для каждой из выделенных пользователем процедур содержит метод для отправки заказа на выполнение этой процедуры. Список выделенных процедур при этом должен декларироваться пользователем в XML файле, причем для каждой процедуры должно быть задано ее расположение, имя генерируемого метода, а также для каждого параметра – тип, имя и направление. Во-вторых, это класс ServConnector, содержащий средства для работы с расположенными на сервере файлами и добавления записей в лог-файл, ведущийся на сервере. В-третьих, это класс DataHandler, от которого должны быть порождены все классы,

использующиеся в качестве типов параметров процедур. Пользователю доступны две группы методов этого класса:

- Абстрактные методы для сериализации и восстановления состояния. Для передачи параметров процедур не используется стандартный механизм сериализации, поскольку при передаче выходных параметров их значения нужно вносить в уже существующие объекты, в то время как существующий механизм позволяет лишь создать новый объект. Поэтому создаваемые пользователем классы должны реализовывать эти методы.
- Методы preRead(), preChange(), preReplace(), которые должны вызываться перед обращением к данным, частичным изменением данных и полной заменой содержимого объекта соответственно. Для потомков этого класса вводится правило, что они должны предоставлять доступ к содержащимся в них данным только через свои методы, в которых вызываются эти три унаследованных метода. Методы preRead и preChange убеждаются, что в объекте есть данные, а preReplace сбрасывает флаг о том, что в объекте нет данных, если он был установлен. Введение двух методов preRead и preChange позволяет, если после сериализации данных объекта производился только доступ на чтение, обнаруживать эту ситуацию и не производить второй раз сериализацию.

Блок взаимодействия с пользовательским кодом является связующим звеном между конкретными прикладными алгоритмами и инструментарием. Его задача состоит в том, чтобы совместно с сгенерированными классами, с одной стороны, предоставить выполняющимся заказам возможность отправки заказов и базовый класс для неготовых величин, реализующий их семантику, и, с другой стороны, предоставить остальным частям инструментария доступ к пользовательской задаче как к некоторому «черному ящику», способному по предоставленному описанию заказа выполнить его, связав идентификаторы неготовых значений выходных параметров с идентификаторами готовых значений или другими идентификаторами неготовых значений, или же в случае невозможности завершить выполнение предоставить подробную информацию о сбое. Также данный блок занимается сбором статистики и управлением набором выполняющихся потоков вычислений, предоставляя локальному планировщику необходимую информацию и приостанавливая или запуская выполнение потоков согласно полученным от него указаниям.

С точки зрения исходного кода блок связи с пользовательским кодом представлен классом потока выполнения заказа ExecuterThread, классом вспомогательной функциональности инструментария ServConnector, классом связи с потоками выполнения StubConnector и базовым классом для структур данных DataHandler, размещенными в пакете org.parallel.client. Класс потока выполнения отвечает за получение сведений о заказе и его выполнение, а класс связи содержит методы, используемые для отправки заказов, получения данных по требованию, а также некоторые средства управления потоками выполнения, которые используются планировщиком.

С блоком взаимодействия с пользовательским кодом тесно связан генератор исходного кода служеюных классов, размещенный в пакете org.parallel.sourcegen. Его работа сводится к чтению предоставленного XML файла с описанием требуемых методов для отправки заказов, проверке его корректности (проверка наличия упоминаемых в файле классов и методов не производится, чтобы не усложнять разработку), генерации исходного кода классов библиотеки, локальной библиотеки, удаленной библиотеки и

исполнителя. Проверка наличия упоминаемых классов не производится, так как они, в свою очередь, могут использовать классы библиотеки и такая проверка привела бы к кольцевой зависимости и усложняла компиляцию приложения. Назначения этих служебных классов таковы:

- Класс библиотеки содержит абстрактные методы для отправки заказов, которые могут использоваться методами выполнения заказов. Для того, чтобы получить доступ к классу библиотеки, метод выполнения заказа должен принимать его как параметр и информация об этом должна быть приведена в файле описания пользовательской задачи. При этом генератор будет записывать в исходном коде вызовы методов с передачей этого параметра.
- Класс локальной библиотеки – реализация класса библиотеки, в которой каждый из методов реализован как локальный вызов соответствующего метода. Этот класс не используется в процессе выполнения пользовательской программы, но может быть использован для локальной отладки создаваемых приложений.
- Класс удаленной библиотеки – реализация класса библиотеки, в которой каждый из методов реализует отправку соответствующего заказа средствами инструментария.
- Класс исполнителя содержит методы, используемые инструментарием в процессе выполнения пользовательской программы.

Блок взаимодействия с пользовательским кодом совместно со сгенерированными классами предоставляет пользовательскому коду операцию отправки заказа, а также реализует операции выполнения заказа и получения значения. Алгоритмы этих операций представлены ниже:

```
Алгоритм выполнения заказа:
получить заказ с сервера;
для (каждого параметра процедуры) {
  если (параметр - входной или транзитный) {
    если (значение параметра известно) {
      установить значение параметра из данных о заказе;
    } иначе {
      связать параметр с идентификатором из данных о заказе;
    }
  } иначе {
    установить значение по умолчанию;
  }
}
выполнить соответствующую процедуру;
для (каждого выходного или транзитного параметра процедуры) {
  отправить на сервер значение параметра;
}
уведомить планировщик об освобождении процессора;
Алгоритм отправки заказа:
отправить на сервер номер процедуры, которую нужно выполнить;
для (каждого входного или транзитного параметра) {
  если (значение параметра известно) {
    отправить значение параметра на сервер;
  } иначе {
    отправить на сервер связанный с параметром идентификатор;
  }
}
получить с сервера набор идентификаторов;
для (каждого выходного или транзитного параметра) {
```

```
  связать параметр с очередным идентификатором;
}
уведомить планировщик о появлении заказа;
Алгоритм получения значения:
получить идентификатор, связанный со значением;
запросить значение с сервера по идентификатору;
если (значение еще не вычислено) {
  сообщить идентификатор планировщику;
  уведомить планировщик об освобождении процессора;
  приостановить работу до указания от планировщика;
  запросить значение с сервера по идентификатору;
}
сбросить связь значения с идентификатором;
```

Планировщик. Планировщик, по сравнению с остальными частями инструментария, содержит достаточно сложную логику. Он реализован классом org.parallel.client.Scheduler. Реализация планировщика основана на относительно простой эвристике наивного планирования, предусматривающей преимущество выполняющихся заказов над еще не стартовавшими. Каждый раз, когда общее количество процессов выполнения заказов оказывается меньше количества процессоров в компьютере, планировщик сначала пытается заполнить процессоры путем продолжения выполнения ранее приостановленных заказов, а затем, если это не удается, загружает новые заказы с сервера. Если это сделать не удается, планировщик начинает периодически опрашивать сервер, загружая появляющиеся задания (подобный опрос сервера планируется в последующих версиях инструментария заменить получением уведомлений от сервера). Затем, когда количество выполняющихся заказов сравняется с количеством процессоров, опрос сервера прекращается.

Планировщик вызывается в двух ситуациях: в случае завершения выполнения заказа и перед началом ожидания данных. Отметим, что при работе планировщика, вызываемого перед началом ожидания данных, нужно учитывать то обстоятельство, что планировщик вызывается из потока выполнения заказа. Это означает, что возможна ситуация, в которой планировщик может попытаться произвести некоторые действия над текущим потоком – например, попытаться его разбудить еще до начала ожидания данных. Случай, когда планировщик вызывается после завершения выполнения заказа, подобных проблем не вызывает, поскольку никаких действий над текущим заказом планировщик произвести не может.

Диаграмма состояний заказа была представлена на рис. 7:

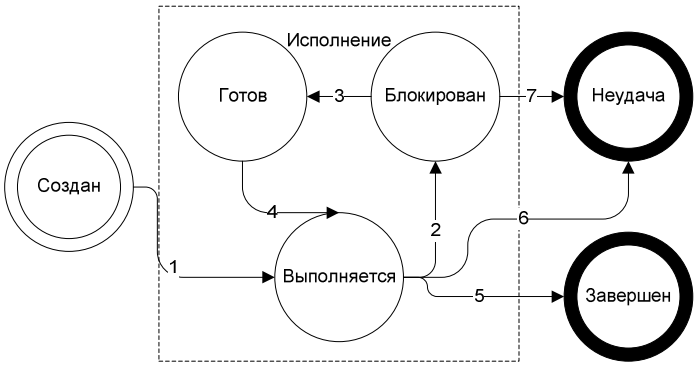

Рис. 7. Диаграмма состояний заказа

Из шести представленных состояний в процессе своего выполнения на клиентском компьютере заказ может быть только в одном из трех состояний: готов, блокирован, выполняется. Для эффективного управления заказами в процессе выполнения выполняющиеся и готовые заказы собраны в списки, а ожидающие данных – в ассоциативный массив, в котором идентификаторам неготовых значений поставлены в соответствие списки заказов, их ожидающие.

Блок кеширования. Блок кеширования данных в клиентской части, представленный классом ProblemServiceProxy пакета org.parallel.client, хранит полученные от сервера данные о значениях, связанных с идентификаторами значений, а также о связях идентификаторов неготовых значений с идентификаторами значений и друг с другом. Для хранения значений, связанных с идентификаторами значений, используется механизм мягких ссылок. Данный класс является реализацией удаленного интерфейса ProblemService. То, что все запросы к серверу проходят через этот блок, позволяет не получать повторно уже имеющиеся на клиенте данные, что уменьшает нагрузку на сеть.

API сервера. Взаимодействие между клиентом и сервером организовано с использованием технологии RMI. Все классы, используемые для удаленных вызовов, собраны в пакете org.parallel.protocol. Удаленные методы собраны в два интерфейса – InfoService и ProblemService. Интерфейс InfoService содержит методы, используемые для настройки клиентской части инструментария в начале процесса вычислений и используемые программой администрирования. Интерфейс ProblemService содержит методы, используемые в процессе вычислений – методы для отправки заказов, получения заказов, отправки и получения данных, получения и отправки сведений о связях идентификаторов, отладочные и другие методы. Также пакет org.parallel.protocol содержит некоторые используемые в описанных двух интерфейсах вспомогательные классы – перечисления для состояния заказа и направления использования параметра, а также класс описания заказа.

Блок обработки запросов. Блок обработки запросов клиентов, представленный классами InfoServiceImpl, ProblemServiceImpl и Problem пакета org.parallel.server, реализует методы, используемые в процессе решения задач – такие, например, как метод для получения идентификатора значения по идентификатору неготового значения, метод для отправки заказа, метод для записи отладочного сообщения в расположенный на сервере файл. Практически каждый метод, используемый в качестве удаленного метода, сводится к выполнению соответствующей операции с базой данных – например, к установке связи между идентификаторами, установке статуса заказа или получению данных по идентификатору. Исключение составляют методы, используемые для сохранения отладочной информации на сервере, и методы для доступа к содержащимся на сервере файлам.

Подсистема управления сущностями. Классы, реализующие функциональность по работе с базой данных, собраны в пакете org.parallel.server.database. Базовым классом в данной функциональности является GenericConnector. Для класса GenericConnector реализован потомок, работающий с базой данных с данными одной задачи – ProblemConnectorImpl. Для того, чтобы обеспечить возможность хранения промежуточных результатов в памяти без использования внешней базы данных, из данного класса выделен интерфейс ProblemConnector, для которого созданы еще две реализации – ProblemConnectorProxy и ProblemConnectorMemoryImpl. Первая из них

использует для хранения данных память, а вторая является прослойкой, добавляющей кеширование на основе механизма мягких ссылок к реализации на основе внешней базы данных. Доступ к базе данных реализован при помощи библиотеки JDBC (Java Database Connectivity).

3. Использование разработанного инструментария

Дистрибутив инструментария состоит из следующих файлов:

- parallel.jar – файл с классами инструментария
- client.bat – файл запуска клиентской части инструментария
- server.bat – файл запуска серверной части инструментария
- problem.bat – файл, используемый для отправки пользовательских заказов
- sourcegen.bat – файл запуска генератора исходного кода
- serverconf.properties – файл с настройками сервера
- etc\log4j\server.properties, etc\log4j\client.properties – файлы с настройками ведения журнала клиентской и серверной частями
- lib\log4j-1.2.13.jar – файл библиотеки Log4J
- work\ - пустой каталог для входных и выходных файлов пользовательских заказов

Среди классов инструментария есть два класса, предназначенные для использования в пользовательском коде. Это уже упоминавшиеся DataHandler и ServConnector. Кроме того, в пользовательском коде могут использоваться сгенерированные с использованием инструментария классы библиотеки. Рассмотрим процесс их генерации подробнее.

Необходимо создать XML файл следующего вида:

```
<library>
    <type fullname="имя_класса_с_именем_пакета"/>
    ....
    <type fullname="имя_класса_с_именем_пакета"/>

    <method name="короткое_имя_метода"
            realname="имя_метода_с_именем_класса_и_пакета"
            [usesLibrary="no"]>
        <param direction="направление" type="имя_класса"
            name="имя_параметра"/>
        ....
        <param direction="направление" type="имя_класса"
            name="имя_параметра"/>
    </method>

    ....

    <method ....
    </method>

    <config basedir="имя_каталога"
            library="имя_класса_с_именем_пакета"
            local="имя_класса_с_именем_пакета"
            remote="имя_класса_с_именем_пакета"
            executer="имя_класса_с_именем_пакета"/>

</library>
```

В этом файле нужно задекларировать все классы, использующиеся в качестве типов параметров процедур (<type>) и все методы. Для каждого метода нужно указать имя этого метода (атрибут realname), имя метода, который нужно сгенерировать (атрибут name) и необязательный атрибут usesLibrary. Если значение атрибута usesLibrary установлено в no, то генерируемые классы будут создаваться исходя из предположения, что данный метод имеет описанную сигнатуру; если же параметр имеет другое значение или не указан, то считается, что у метода есть еще один первый параметр, тип которого – сгенерированный класс библиотеки. Для каждого из методов в качестве вложенных элементов указаны все формальные параметры, причем для каждого указан тип, имя (которое используется в качестве имени параметра в генерируемом классе библиотеки) и направление параметра (входной, выходной или транзитный, I/O/IO). Кроме того, нужно разместить в файле элемент <config>, указав в его атрибутах имя каталога, где будет размещена программа, а также имена генерируемых классов.

Чтобы на основе созданного файла сгенерировать исходный код требуемых классов, нужно запустить файл sourcegen.bat, передав ему в качестве параметра имя созданного файла. В результате запуска либо будет выдано сообщение об успешной генерации файлов, либо сообщение об ошибке.

Подготовка к запуску созданного параллельного приложения состоит из нескольких этапов:

1. Нужно скопировать созданные пользовательские классы в папку с дистрибутивом инструментария.
2. Если при работе приложения будет использоваться внешняя база данных, то соответствующий JDBC драйвер нужно скопировать в папку lib сервера.
3. В файлах client.bat и problem.bat к описанным в них значениям CLASSPATH нужно добавить путь к пользовательским классам, а в файле server.bat – путь к скопированному JDBC драйверу.
4. В файлах client.bat и problem.bat нужно указать адрес или имя компьютера, который будет работать как сервер вычислений.
5. Модифицированный таким образом дистрибутив инструментария нужно распространить по компьютерам кластера.
6. На сервере нужно также заполнить файл serverconf.properties. Ключи в этом файле могут быть следующими:
    - dbClass, dbUrl – имя класса JDBC драйвера и URL для подключения к базе данных (если не указаны, внешняя база данных не используется).
    - dbUser, dbPassword – имя пользователя и пароль для базы данных (если не указаны, подключение производится без аутентификации).
    - recreateDB – если такой ключ существует в properties-файле, то производится создание или пересоздание всех необходимых таблиц во внешней базе данных. В этом случае должны также быть указаны значения для ключей maxNumInputParams и maxNumOutputParams, показывающие максимально возможное количество входных и выходных параметров процедуры соответственно.
    - startServer – если такой ключ существует, производится запуск сервера. В этом случае должно быть указано значение ключа executerClass, показывающее имя сгенерированного класса исполнителя.

7. Если программа использует какие-либо входные файлы, их нужно разместить в каталоге work\ сервера.
8. Процесс запуска параллельного приложения состоит из следующих этапов:
9. На сервере нужно запустить файл server.bat
10. На всех компьютерах, используемых в качестве клиентских, нужно запустить файл client.bat. Поскольку на серверную часть инструментария возложены функции, не связанные с высокой вычислительной нагрузкой, на компьютере, на котором работает сервер, можно также запустить и клиентскую программу.
11. Запустить на любом из клиентских компьютеров файл problem.bat, указав в качестве параметра имя метода (в формате «имя_пакета.имя_класса.имя_метода»), тип единственного параметра которого есть сгенерированный класс библиотеки. Этот метод должен в ходе своего выполнения отправить все необходимые пользовательские заказы.
12. Дождаться завершения выполнения программы. Во время выполнения программы клиенты выводят в консоль информацию об изменениях в количестве использованных процессоров, а сервер – часть этой информации с клиентов и уведомления в моменты времени, когда оказываются выполненными все заказы.
13. Если с использованием тех же классов нужно решить еще одну или несколько задач – перейти к этапу 3. Иначе остановить серверную и клиентские программы.

Для примера рассмотрим задачу исследования информативности формируемых диагностических признаков с помощью процедуры полного перебора. Задача заключается в сравнении вероятностей правильного распознавания, которую способны обеспечить квадратичные разрешающие правила, построенные на основе данных по всем непустым поднаборам признаков, и нахождении минимального набора среди наборов, обеспечивающих максимальную вероятность правильного распознавания. Последовательный алгоритм решения данной задачи выглядит следующим образом:

```
найти математические ожидания значений признаков для каждого из
классов и построить ковариационные матрицы;
для (всякого непустого поднабора признаков) {
  построить квадратичное разрешающее правило и вычислить его значения
на предоставленных значениях наборов признаков для объектов каждого
их классов;
  найти максимальную вероятность правильного распознавания, которую
может обеспечить построенное правило;
  на первом витке цикла сохранить полученный результат как наилучший,
а на всех последующих сделать это в случае, если он лучше ранее
полученного результата.
}
```

Как видно, данный алгоритм не содержит существенных отрезков времени между построением данных и их использованием, поэтому несколько изменим его. Будем производить сравнение результатов анализа наборов признаков не по мере анализа этих наборов, а после того, как анализ всех наборов окончен – и тогда, очевидно, анализ отдельного набора можно будет выделить в отдельный заказ. Чтобы избежать высоких накладных расходов на передачу данных, будем

выносить в один заказ анализ не одного набора признаков, а их большего количества. В этом случае алгоритм примет такой вид:

```
найти математические ожидания значений признаков для каждого из
классов и построить ковариационные матрицы;
для (всякой группы поднаборов признаков) {
  найти наилучший из наборов признаков из этой группы (как в
предыдущем алгоритме);
}
найти лучший из полученных результатов;
```

На примере данного алгоритма можно рассмотреть еще один вопрос, связанный с областью применимости предлагаемого инструментария. Легко видеть, что предложенный алгоритм легко можно реализовать как с применением параллелизма заданий, так и с применением параллелизма данных. Реализация с применением параллелизма данных и библиотеки MPI будет выглядеть следующим образом:

```
если (текущий процессор - нулевой) {
  вычислить математические ожидания значений признаков для каждого из
классов, построить ковариационные матрицы и передать вычисленные и
исходные данные остальным процессорам;
} иначе {
  получить эти данные от нулевого процессора;
}
провести вычисления над своей частью набора признаков;
если (текущий процессор - нулевой) {
  собрать данные с остальных процессоров и найти лучший набор среди
найденных всеми процессорами
} иначе {
  отправить свой лучший набор на нулевой процессор;
}
```

Пункт «провести вычисления над своей частью набора признаков» этого алгоритма неявно содержит предположение, что работу можно разделить между всеми процессорами таким образом, чтобы время работы каждого из процессоров было одинаковым. Если кластер однороден (достаточно распространенный случай), работу просто можно разделить на N равных частей, где N – количество процессоров. В случае же неоднородного кластера решение этой задачи уже не столь тривиально: нужно либо распределять работу с учетом тактовых скоростей процессоров (что достаточно неудобно), либо так или иначе моделировать работу предыдущего алгоритма, когда работа разделяется на большее количество частей, которые затем выделяются нулевым процессором остальным по требованиям от них. Такой вариант связан еще и с вопросом о том, что обработку запросов нужно проводить одновременно с вычислениями, проводимыми на нулевом процессоре.

Аналогичные вопросы возникают и в случае, если кластер однороден, но неоднородны задания, которые должны выполняться параллельно. В этом случае можно или разделить процессоры на группы, поручив каждой из групп выполнение некоторой достаточно однородной части заданий и сделав размеры групп приблизительно пропорциональными объему вычислений в группах (что может быть сложно во многих случаях), или разделив задания на группы, и выполняя поочередно группы заданий на всех доступных процессорах. Таким образом, опять возникает ситуация, в которой параллелизм данных либо не

имеет существенных преимуществ перед параллелизмом заданий, либо дает меньшую эффективность. Это и есть те алгоритмы, для которых применение предлагаемой технологии позволяет получить значительный выигрыш. Алгоритмами, для которых использование этой технологии нецелесообразно, являются те алгоритмы, которые невозможно эффективно реализовать, используя лишь те операции для взаимодействия процессоров, которые предоставляет рассматриваемая технология.

Рассмотрим реализацию рассматриваемого алгоритма в рамках разработанного инструментария. Для простоты будем считать, что уже реализован его последовательный вариант и рассмотрим те изменения, которые в него нужно внести.

Пусть реализован класс identification.Algorithm следующего вида:

```
class Algorithm {
    public static void main(String args[]) {
        //Чтение входных данных из файла
        //Сравнение наборов признаков
        //Запись результата в файл
    }
}
```

Объединим анализируемые наборы в группы, как было описано в предыдущем пункте. В этом случае класс примет следующий вид:

```
class Algorithm {

    public static void analyzeGroup(.....) {
        //Анализ группы наборов признаков
    }

    public static void main(String args[]) {
        //Чтение входных данных из файла
        //Сравнение наборов признаков
        //Запись результата в файл
    }
}
```

Чтобы выполнение метода analyzeGroup можно было вынести в отдельный заказ, необходимо, чтобы все его параметры были потомками класса DataHandler. Поэтому реализуем потомки этого класса для всех структур данных, которые нужно будет использовать в качестве параметра – для матрицы, вектора, целочисленного вектора, целого и дробного числа. Затем заменим типы параметров метода analyzeGroup класса Algorithm созданными классами и уберем неиспользуемый параметр метода main.

Далее необходимо сгенерировать методы для отправки заказов. Для этого нужно описать процедуры во внешнем XML файле, который для данной задачи будет выглядеть следующим образом:

```
<library>
    <type fullname="identification.DoubleNum"/>
    <type fullname="identification.IntNum"/>
    <type fullname="identification.Matrix"/>
    <type fullname="identification.Vertex"/>
    <type fullname="identification.IntVertex"/>
```

```
    <method name="analyzeGroup" usesLibrary="no"
          realname="identification.Algorithm.analyzeGroup">
        <param direction="I" type="Matrix" name="pnts0"/>
        <param direction="I" type="Matrix" name="pnts1"/>
        <param direction="I" type="Vertex" name="mid0"/>
        <param direction="I" type="Vertex" name="mid1"/>
        <param direction="I" type="Matrix" name="covar0"/>
        <param direction="I" type="Matrix" name="covar1"/>
        <param direction="I" type="IntVertex" name="indices"/>
        <param            direction="O"            type="IntVertex"
name="bestCombination"/>
        <param direction="O" type="DoubleNum" name="lambdaVal"/>
        <param direction="O" type="IntNum" name="numErrors0"/>
        <param direction="O" type="IntNum" name="numErrors1"/>
        <param direction="I" type="IntNum" name="indicesMiddle"/>
    </method>

    <method name="main" realname="identification.Algorithm.main"/>

    <config basedir="c:\problem\src\"
            library="identification.Library"
            local="identification.LocalLib"
            remote="identification.RemoteLib"
            executer="identification.Executer"/>

</library>
```

Запустим генератор классов, входящий в состав инструментария, передав ему в качестве параметра имя созданного XML файла. В результате будут сгенерированы описанные четыре класса.

Добавим еще один параметр к методу main и заменим вызов метода для анализа группы наборов признаков соответствующим методом библиотеки. Параллельная программа готова.

Рассмотрим более подробно процесс выполнения полученной параллельной программы. Сервер после запуска сначала выполняет пересоздание базы данных, если это необходимо, после чего создает новые удаленные объекты (в терминах технологии RMI), через которые к серверу будут обращаться клиенты. Больше никаких действий он не производит. Каждый из клиентов в момент старта получает доступ к этим объектам, через которые получает имя сгенерированного класса исполнителя. После этого проверяется корректность классов задачи. Проводятся следующие проверки:

Каждый из классов, используемых в качестве параметров процедур, должен быть корректным. Под корректностью метода понимается, что метод должен существовать, быть не абстрактным потомком класса DataHandler и иметь конструктор по умолчанию с модификатором доступа public.

Каждый из выделенных методов должен быть корректным. Под корректностью метода понимается, что метод должен существовать, быть статическим, иметь требуемую сигнатуру и возвращаемый тип void.

Подобные проверки позволяют еще в момент старта клиентской части обнаружить некоторые ошибки в пользовательском коде, которые могут возникать при неаккуратном переносе пользовательского кода на клиентские компьютеры. После проведения подобной проверки клиент начинает свою работу с запуска планировщика, который пытается загрузить на выполнение количество заказов, равное количеству процессоров компьютера. Если это не

удается, то планировщик после этого периодически опрашивает сервер и при появлении новых заказов загружает их и начинает выполнение.

Для экспериментальной проверки эффективности разработанного инструментария был проведен эксперимент для задачи исследования информативности формируемых диагностических признаков с помощью процедуры полного перебора [6], количества признаков 23, 24 и 25 и 1, 2, 3, 5 и 10 компьютеров с процессорами Inter Pentium 1.7 GHz и 256 mb RAM, соединенных при помощи Fast Ethernet. Эксперимент показал достаточно высокую эффективность предлагаемого подхода: такая величина, как произведение отрезка времени между запуском задачи и получением ее решения на количество компьютеров, при переходе от 1 к 2, 3 и 5 компьютерам возрастает не более чем на 1,13%, а при переходе к 10 компьютерам – не более чем на 3,25%. График зависимости количества времени, потраченного на вычисления, от количества компьютеров и размерности задачи представлен на рис. 8:

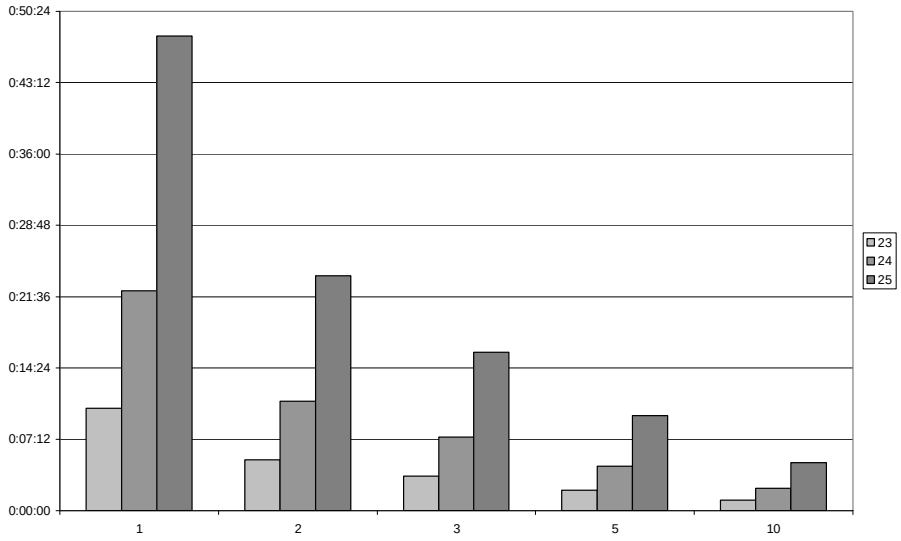

Рис. 8. Зависимость времени, затрачиваемого на решение тестовой задачи, от размерности задачи и количества компьютеров в составе кластера

Из приведенной диаграммы (рис. 8) видно, что произведение количества компьютеров на затраченное время остается величиной примерно постоянной. С теоретической точки зрения эта величина должна постепенно увеличиваться при увеличении размера кластера из-за увеличения простоев и расходов времени на передачу данных. В проведенном эксперименте эта величина возрастает не более чем на 1,13% при переходе от одного к двум, трем и пяти компьютерам, и не более чем на 3,25% при переходе от одного к десяти компьютерам в кластере.

4. Заключение

В данной статье описана технология неявного распараллеливания кластерных вычислений, основанного на заказах, которая позволяет для

алгоритмов, реализуемых с использованием параллелизма заданий, достаточно легко переходить от существующих последовательных реализаций к параллельным реализациям, внося незначительные изменения как в код, так и в логику работы алгоритма прикладной задачи.

Предложенная технология реализована в виде инструментария на основе средств языка программирования Java. Описан реализующий ее инструментарий с точки зрения системного разработчика и с точки зрения прикладного программиста. Рассмотрены основные вопросы, возникающие как при разработке инструментария, так и при его практическом использовании. Эффективность разработанной технологии подтверждена экспериментально решением тестовой задачи.

Разработан метод вычисления времени выполнения произвольной прикладной задачи на однородном достаточно большом кластере при использовании предложенной технологии распараллеливания.

Список литературы

1. Тхань Фыонг Нгуен, Шелестов А.Ю. Параллельная реализация алгоритмов фильтрации космических изображений. // Проблемы управления и информатики. – 2005. – № 5. – С. 121-132
2. Абрамов С.М., Московский А.А., Роганов В.А., Шевчук Ю.В., Шевчук Е.В., Парамонов Н.Н., Чиж О.П.. Open TS: архитектура и реализация среды для динамического распараллеливания вычислений. // Научный сервис в сети Интернет: технологии распределенных вычислений: Труды Всероссийской научной конференции, 19 – 24 сентября 2005, г. Новороссийск. – М.: Изд-во МГУ. – С. 79 – 81.
3. Roganov V., Moskovsky A., Abramov S. The Open TS parallel programming system. // The Twelfth International Conference on Parallel and Distributed Systems, Minneapolis, USA (ICPADS, July 12-15, 2006). – http://skif.pereslavl.ru/skif/index.cgi?module=chap&action=getpage &data=publications\pub2006\opents_ext.doc
4. Khoroshevsky V.G., Mamoilenko S.N., Maidanov Y.S., Sedelnikov M.S. Space-distributed multicluster computer system for parallel multiprogramme regimes modeling. // Сборник трудов конференции «МОДЕЛИРОВАНИЕ-2006», 16-18 мая 2006, Киев, ИПМЭ им. Г.Е. Пухова НАНУ. – С. 67 – 69.
5. Kolding T. E., Larsen T. High Order Volterra Series Analysis Using Parallel Computing. – http://citeseer.ist.psu.edu/242948.html.
6. Павленко В.Д., Фомин А.А., Череватый В.В. Построение пространства диагностических признаков на основе моделей объектов контроля в виде рядов Вольтерра // Вторая международная конференция по проблемам управления. Москва, Институт проблем управления им. В.А.Трапезникова РАН, 17-19 июня 2003.: Избранные труды в двух томах. – М.: Институт проблем управления, 2003. – Том 2. – С. 110 – 117.
7. Павленко В.Д., Бурдейный В.В. Принципы организации кластерных вычислений с помощью неявного распараллеливания, основанного на заказах // Труды III Международной конференции «Параллельные вычисления и задачи управления» PACO ‘2006 памяти И.В. Прангишвили. Москва, 2-4 октября 2006 г., Институт проблем управления им. В.А. Трапезникова РАН. М.: Институт проблем управления им. В.А. Трапезникова, 2006. – С. 670 – 690, CD ISBN 5-201-14990-1
8. Pavlenko V., Burdeinyi V. Computing Simulation in Orders Based Transparent Parallelizing. – ICIM 2008: Proceedings 2nd International Conference on Inductive Modelling, September 15-19, 2008, Kyiv, Ukraine, pp.168-171. – ISBN 978-966-02-4889-2.
9. Антонов А.С. Параллельное программирование с использованием технологии MPI. – М.: Изд-во МГУ, 2004. – 71 с.
10. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. – СПб.: БХВ–Петербург, 2002. – 608 с.