РЕШЕНИЕ ПРОБЛЕМЫ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ НА ОСНОВЕ ПОНЯТИЙ «ПРОСТРАНСТВО-ВРЕМЯ»

А.И.Илюшин, М.А.Оленин, С.А.Васильев

Кафедра вычислительной механики, механико-математический факультет МГУ
Институт прикладной математики ИПМ РАН

Аннотация

Рассматривается построение распределенных систем путем композиции программных объектов. Предлагается задавать топологию связей между объектами, описывая «окрестность» каждого объекта в виде списка «формальных соседей». Синхронизацию эволюции объекта и его «соседей» предлагается описывать с помощью «локального времени» объекта и его окрестности. Приводятся результаты программирования и счета реальных задач на суперкомпьютере, показывающие возможность сведения трудоемкости создания распределенных программных систем к трудоемкости локального программирования.

Ключевые слова и выражения:
Композиция программных объектов, топология связей объектов, формальные и фактические соседи, синхронизация эволюции объектов, локальное время объекта и его окрестности.

1. Введение

Общепризнанно, что в настоящее время программирование многопроцессорных (многоядерных) вычислительных систем является задачей, которая требует от программиста и высокой квалификации, и больших трудозатрат. В частности, можно привести цитату из интервью ветерана в области вычислительной техники Джона Хэннеси, президента Стэндфордского университета [1] :

"*... когда мы начинаем говорить о параллелизме и легкости использования действительно параллельных компьютеров, мы говорим о проблеме, которая труднее любой проблемы, с которой встречалась наука о компьютерах <…> Я бы запаниковал, если бы я работал в промышленности.*"

Попытаемся разобраться, в чем же конкретно заключается проблема. Представим себе программную систему, состоящую из множества взаимодействующих программ, распределенных по множеству процессоров. В этом случае в отличие от одной последовательной программы, выполняющейся на одном процессоре, возникают две основные проблемы:

1. Проблема установления связей между удаленными друг от друга программами и управления этими связями.
2. Проблема частичного упорядочения действий, выполняемых этими программами, и синхронизации их взаимодействия.

Именно для решения этих проблем в «ручном» режиме в настоящее время требуются и незаурядные умственные способности, и большие трудозатраты [2]. Оказывается, что параллельный случай можно свести к последовательному в смысле требований к

программисту и процессу программирования путем ввода принципиально новых программных механизмов, основанных на понятиях «пространство-время».

Дело в том, что ***любая*** система состоит из некоторого множества частей с некоторым конкретным набором связей между ними. Это означает, что практически все прикладные программисты будут программировать примерно одинаковые действия для управления связями между частями программной системы. Аналогичная ситуация с арифметическими выражениями существовала, например, до появления Фортрана. Каждый программист программировал в Ассемблере арифметические выражения. Вынесение «за скобки» в системную часть повторяемых всеми действий – это естественный выход из такого рода ситуаций.

Такое вынесение может быть сделано только на основе формальной фиксации некоторого понятия, описывающего алгоритмизируемый класс действий. Для множеств, в которых определены связи между элементами (в другой формулировке для каждого элемента определена окрестность в виде подмножества элементов, связанных с данным), таким понятием является понятие топологического пространства.

Упорядочение действий, выполняющихся в различных параллельно эволюционирующих частях системы, с точки зрения авторов, адекватно описывается понятием времени.

Именно эти два понятия и были положены в основу при разработке комплекса программных средств, целью которого является сведение трудоемкости программирования параллельных систем к трудоемкости последовательного программирования.

Практической основой для представляемой разработки являлось решение реальных задач механики на современных суперкомпьютерах. В частности было просчитано несколько вычислительных моделей из области газовой динамики. Однако авторы считают, что в предлагаемых средствах программирования формализованы понятия, общие для любых прикладных систем.

2. Основные идеи решения

2.1. Декомпозиция/композиция программной системы

Начнем с общей схемы создания распределенной программной системы, как она представляется авторам. При проектировании любой программной системы естественно начать с определения ее характеристик как единого целого. Если проектируется именно распределенная система, то далее необходимо провести декомпозицию системы, выделить части, которые предположительно будут выполняться на отдельных процессорах. Например, в случае создания вычислительных моделей для физических областей такая декомпозиция обычно выполняется на уровне дискретной модели [3]. Затем следует этап программирования выделенных частей.

Предполагается, что на этапе декомпозиции для каждой части был определен ее интерфейс с другими частями в виде набора функций, выполняемых данной частью, и наборов функций, которые выполняются другими частями и используются данной. После этого

программирование каждой части может быть выполнено независимо от других частей системы.

И, наконец, нужно сделать композицию, сборку целевой системы из частей в единое целое. Основная цель представляемой авторами работы – разработка системных программных средств, позволяющих с малыми трудозатратами запрограммировать сборку независимо созданных программных подсистем в единую распределенную систему.

2.2. Объектно-ориентированный подход

В данной работе за основу берется обычная объектно-ориентированную модель программирования, которая дополняется некоторыми новыми чертами. Программная система строится из множества взаимодействующих друг с другом объектов в рамках некоторой объектно-ориентированной среды программирования. Используется схема, при которой в один объект может одновременно войти несколько процессов и один процесс может пройти через много объектов, возможно расположенных на разных процессорах.

Предполагается, что разбиение системы на объекты выполняется либо на этапе проектирования при создании сложных неоднородных систем, либо может быть сделано автоматически. В качестве систем, поддающихся автоматической декомпозиции, можно, например, привести вычислительные модели задач газовой динамики для областей, сравнительно простых с точки зрения геометрии и протекающих в них процессов.

Для каждого объекта, как обычно, определен **интерфейс** — набор функций для вызовов. Взаимодействие объектов может производиться только вызовами одним объектом операций из интерфейса в другом объекте.

Во время счета множество входящих в модель объектов отображается на множество процессоров. Типичный случаем будет считать выделения одного процессора (возможно виртуального) для одного объекта.

2.3. Формальные/фактические соседи

Для того чтобы множество объектов объединить в систему, необходимо задать связи между ними. Используем для этого понятие «соседства». Объект Б считается соседом объекта А, если в процессе эволюции объекта А требуется взаимодействие с объектом Б. Стоит заметить, что понятие соседства может быть как односторонним, так и двухсторонним (объектам требуется взаимодействие друг с другом).

Множество всех соседей конкретного объекта определяет «внешний мир» этого объекта, с которым может происходить его взаимодействие.

Напомним, что для каждого объекта определен некий интерфейс и взаимодействие происходит только с помощью вызовов функций из этого интерфейса. Поэтому, при программировании конкретного класса программисту для описания взаимодействия с другими объектами в модели достаточно иметь список «формальных соседей» с их интерфейсами. «Фактические» соседи, классы которых реализуют интерфейсы из описаний формальных соседей, появятся только во время счета.

Таким образом, при программировании каждого объекта прикладной программист может исходить только из локальных соображений о «внешнем мире» этого объекта, представленном в виде списка формальных соседей.

2.4. Топологические пространства объектов и организация связей между объектами

Распределенная система строится из частей, которые взаимодействуют друг с другом. В программной реализации связи между частями задаются ссылками. В традиционном программировании необходимо вручную определить все ссылки для каждой из частей системы. Если частей десятки тысяч со сложной топологией связей - это очень трудоемкая задача. Для объектно-ориентированного случая — это программирование прикладным программистом присваивания локальным переменным объекта ссылок на соседние объекты.

Кроме трудоемкости «ручного» формирования ссылок возникает системная проблема коррекции ссылок в случае перемещения объектов из одного вычислительного узла в другой. Это бывает необходимо для балансировки нагрузки многопроцессорной вычислительной системы (МВС) путем «подкачки/выталкивания» неактивных или низкоприоритетных объектов (виртуальная память объектов - аналог подкачки страниц для традиционной виртуальной памяти). При «ручном» управлении ссылками решение этой задачи становится слишком сложным. В нашем случае использование списка формальных соседей и вынесение задачи формирования ссылок на фактических соседей из ведома прикладного программиста в системную часть позволяет достаточно просто решить эту проблему.

Для простого и наглядного задания связей между объектами введем понятие — «пространство объектов». С точки зрения математика «пространство объектов» — это обычное топологическое пространство. Автоматизация состоит в предоставлении пользователю простого интерфейса для работы с топологией связей, а задачи создания и поддержания актуальности ссылок реализуются механизмами внутри предлагаемой нами системы.

В нашем случае топология определяется с использованием метрического пространства и функции близости. В метрическом пространстве каждому объекту сопоставляется некоторая точка из этого пространства, а функция близости определяет наличие связи между парами точек. Прикладному программисту предлагается либо статически до начала основного счета, либо динамически во время счета задать координаты каждого объекта в используемом метрическом пространстве.

Например, в произвольном графе можно пронумеровать вершины целыми числами и получить размещение в одномерном метрическом пространстве. Для определения функции близости удобно использовать таблицу пар соседних вершин (ребер). Для любой пары номеров вершин — функция отвечает на вопрос — есть ли такая пара в таблице или нет.

Другой пример использования — это задание многомерной решетки. В качестве пространства выбирается многомерное пространство с целочисленными координатами. Для каждой точки легко определяется множество возможных соседей — это точки, у которых ровно одна координата отличается на 1 от соответствующей координаты рассматриваемой точки. Функция близости будет отвечать на вопрос о попадании точки в это множество.

2.5. Синхронизация вызовов на основе локального времени объектов

Синхронизация вычислений — это сложная задача, знакомая каждому, кто сталкивался с многопоточным программированием. Для решения этой проблемы авторами предлагается использовать механизм, использующий понятие «ВРЕМЯ».

«ВРЕМЯ» практически во всех областях деятельности человека всегда использовалось и используется для упорядочения действий. В частности, время в качестве средства синхронизации используется в многочисленных системах моделирования цифровых устройств и дискретных систем управления.

В нашем случае «ВРЕМЯ» – это разметка числами последовательности действий в рассматриваемом объекте (или в других терминах – разметка последовательных состояний объекта). Синхронизация достигается за счет использования механизма, который обеспечивает взаимодействие пары соседних объектов только при равенстве их локальных времен.

При таком подходе прикладному программисту не нужно следить за «соседними» объектами и синхронизировать вычисления в них. Достаточно просто продвигать значение переменной «ВРЕМЯ» *локально* в каждом объекте, а синхронизирует вычисления описываемая система автоматически.

3. Средства программирования прикладных систем

Рассмотрим конкретные средства программирования прикладных распределенных систем, основанные на вышеизложенных идеях. Предполагается, что работа ведется в более или менее любой существующей среде объектно-ориентированного программирования, дополненной некоторым числом дополнительных классов и дополнительной библиотекой поддержки времени счета.

В данной работе все примеры приводятся в контексте языка PYTHON и соответствующей системы программирования. Будем называть эту новую среду с дополнительными средствами для создания распределенных программ системой OST (Object — Space — Time)

Во фрагментах программ, приводимых ниже в качестве примеров, используются следующие обозначения:

- шрифт «*курсив*» указывает на то, что данное имя является абстрактным и в конкретной прикладной программе должно быть заменено программистом на конкретное;

- шрифт «**полужирный**» указывает на то, что данное имя является фиксированным в среде OST и в любой прикладной программе должно быть именно таким;

- шрифт «обычный» указывает на то, что данное имя является фиксированным в среде данного языка программирования.

3.1. Определение топологии связей в прикладной системе

При использовании системы OST задание связей между объектами в прикладной системе производится в виде задания топологии некоторого вспомогательного пространства.

Термины из математики используется здесь потому, что понятия окрестности каждого объекта и связности системы в целом оказываются просто калькой соответствующих определений из математики, а программные средства для определения связей между объектами можно рассматривать как конкретный механизм определения топологического пространства. В результате, оказывается удобно использовать хорошо известный набор топологических понятий.

Рассмотрим конкретные механизмы работы с топологиями множеств объектов в предлагаемой нами системе.

3.1.1. Определение топологии.

В нашей работе под заданием топологии некоторого пространства, мы понимаем задание для каждой точки из этого пространства окрестности в виде множества соседних точек или, проще говоря, соседей. Для объектов, из которых состоит создаваемая распределенная система, понятие окрестности соответствует понятию “внешнего окружения” объекта.

В системе OST топология задаются с помощью класса, в котором определена либо функция близости, проверяющая соседство пары точек, либо функция описания окрестности, задающая множество точек входящих в окрестность к данной. Такое разделение позволяет простым способом описать разные по устройству топологии. Так функция описания окрестности хорошо подходит для описания регулярных структур, например, целочисленных решеток. Функция близости, наоборот, подходит для описания нерегулярных структур, например, в случае взаимодействия через поле тяготения в небесной механике.

3.1.1.1. Задание окрестности объекта с помощью функции близости.

Функция близости для пары точек A и B определяет принадлежность точки B окрестности точки A. Сама окрестность фиксированной точки A может быть построена системой OST, например, путем перебора всевозможных значений B и проверкой с помощью функции близости. Проиллюстрируем задание топологии на примере графа с пронумерованными вершинами. Граф связей, не имеющий никакой регулярной структуры, можно описать только с помощью таблицы, в которой для каждой вершины хранится список соседних вершин. Функция близости в таком случае проверяет наличие ребра между парой вершин в этой таблице, названной в приведенном ниже примере `self.`*`edges`*.

Для того, чтобы при задании топологии имелась возможность хранения информации, описывающей например, ребра графа, функция близости реализуется не как одиночная функция, а помещена в класс описания топологии.

```
class applied_topology(ost.Topology.Abstract):
    #Таблица, задающая ребра графа
    #Для каждой вершины p1 в self.edges[p1] хранится список
    #вершин p2 с которыми p1 связана ребром
```

```
    self.edges = { p1: [p1_1,...,p1_k1]
                   p2: [p2_1,...,p2_k2]
                   ...
                   pM: [pM_1,...,p1_kM]
                 }

    #Функция близости проверяет наличие вершины p2
    #в списке концов ребер, выходящих из p1
    def proximity(self, p1, p2):
        # Оператор in возвращает bool значение вхождения
        # в списке self.edges[p1] элемента p2
        return p2 in self.edges[p1]
```

3.1.1.2. Задание окрестности объекта с помощью функции описания окрестности.

Функция описания окрестности для некоторой точки возвращает в виде списка координаты всех точек, которые входят в её окрестность.

Рассмотрим применение этой функции на том же примере задания графа связей.

```
class applied_topology(ost.Topology.Abstract):
    #Таблица, задающая ребра графа
    #Для каждой вершины p1 в self.edges[p1] хранится список
    #вершин p2 с которыми p1 связана ребром
    self.edges = { p1: [p1_1,...,p1_k1]
                   p2: [p2_1,...,p2_k2]
                   ...
                   pM: [pM_1,...,p1_kM]
                 }

    #Функция описания окрестности вершины p
    #Возвращает список всех вершин, с которыми соединена данная p
    def neighborhood(self, p):
        return self.edges[p]
```

Какую из предложенных функций использовать для задания топологии конкретной распределенной системы остается на усмотрение прикладного программиста.

В некоторых случаях удобнее использовать функцию близости. Для примера рассмотрим двумерную плоскость и окрестности в виде кругов.

```
class applied_topology(ost.Topology.Abstract):
  # Функция вычисления расстояния между точками
  def distance(self, p1, p2):
      return sqrt( (p1[0]— p2[0])**2 +(p1[1] - p2[1])**2)

  # Функция близости. Расстояние между точками меньше, чем radius
  def proximity(self, p1, p2):
      return self.distance(p1, p2) < radius
```

Для других ситуаций, использование функции описания окрестности оказывается удобнее и нагляднее. Например, в случае двумерной целочисленной решетки можно описать множество точек, входящих в окрестность, следующим образом:

```
class applied_topology(ost.Topology.Abstract):
def neighborhood(self, p):
    return [
            [p[0] + 1, p[1]], # точка справа
            [p[0] - 1, p[1]], # точка слева
            [p[0], p[1] + 1], # точка сверху
            [p[0], p[1] — 1]  # точка снизу
           ]
```

Стоит заметить, что системе OST в случае использования функции близости для определения окрестности конкретной точки необходимо произвести проверку на соседство с помощью этой функции для всех точек в паре с данной точкой. Для случая статической топологии связей этот факт не очень существенен. Однако, в случае изменения топологии связей в процессе счета, с целью оптимизации времени работы прикладной системы, рекомендуется использовать функцию описания окрестности.

В системе OST предусмотрено наличие «стандартных» классов топологий, включенных в систему. Эти классы реализуют с помощью вышеописанных механизмов наиболее часто встречающиеся типы топологий. В случае «нестандартных» топологий прикладной программист может создать собственный класс топологии и использовать его в своих прикладных системах.

3.1.2. Локальная и глобальная топология.

В данном разделе мы рассмотрим, как при создании модели использовать имеющиеся классы топологий для задания связей между объектами в прикладной системе. В системе OST при создании модели программист может задать класс топологии как общий для всех объектов в системе, либо его можно задать индивидуально для объектов со специфической топологией окрестности. Система OST при запуске на счет , а в случае динамического изменения связей и во время счета, автоматически определит и установит все необходимые связи между объектами.

Таким образом, экземпляры классов топологий в системе OST могут использоваться двумя способами: как глобальные (задающие топологии всех объектов) и локальные (задающие топологию окрестности одного объекта).

Объект глобальной топологии задается в программе инициализации прикладной системы как параметр **topology** создаваемой модели. Приведем для примера часть из программы, в которой задается двухмерная целочисленная решетка с помощью стандартного класса **ost.Topology.Mesh**, который задает топологию целочисленных решеток.

```
# Определение глобальной топологии
objInit.topology = ost.Topology.Mesh(dimension = 2)
```

Если глобальная топология не определена, то в системе OST используется стандартная топология, в которой окрестность каждого объекта пуста. Тогда все связи должны быть заданы локальными топологиями.

Объект локальной топологии определяется при создании объекта, например, в программе инициализации прикладной системы как параметр **topology** создаваемого объекта. Приведем пример задания окрестности с помощью стандартного класса

ost.Topology.Neighborhood, в котором окрестность точки описывается в виде константного множества координат соседних точек

```
# Определение локальной топологии
object.topology = ost.Topology.Neighborhood([[x1,y1],...,[xN,yN]])
```

В данном примере определяется окрестность объекта *object*, которая состоит из точек [x1,y1],...,[xN,yN].

Для иллюстрации использования локальных и глобальных топологий, приведем пример программы создания модели.

```
# Определение глобальной топологии
objInit.topology = ost.Topology.Mesh(dimension = 2)

# Создание объекта
object1 = objectInit.createObject( <...> )
# помещение объекта в точку с координатами [0,0]
objInit.set(object1, [0,0])

object2 = objectInit.createObject( <...> )
objInit.set(object2, [0,1])

object3 = objectInit.createObject( <...> )
objInit.set(object3, [1,1])

object4 = objectInit.createObject( <...> )
# Определение локальной топологии
object4.topology = ost.Topology.Neighborhood([[0,0]])
objInit.set(object4, [2,2])
```

В этом примере с помощью глобальной топологии задается связь между парами объектов с координатами [0,0], [0,1] и [0,1], [1,1]. С помощью локальной топологии для объекта с координатами [2,2] задается односторонняя связь с объектом [0,0].

Использование других классов топологий, в том числе и пользовательских, аналогично приведенному примеру.

3.1.3. Формальные и фактические соседи.

В данном разделе рассмотрим механизм взаимодействия объектов прикладной системы со своими окрестностями. При создании модели с помощью системы OST пользователь описывает некоторый набор типов объектов. Тип определяется прикладным классом, исходя только из локальных соображений, относящихся только к объектам описываемого типа.

Конкретно локальность здесь означает, что программист вполне традиционным образом описывает внутреннее функционирование объекта данного типа, а «окружение» определяется в виде списка «формальных соседей».

Понятие «список формальных соседей» можно рассматривать как существенное обобщение понятия списка формальных параметров для подпрограммы. Элементы этого списка - это объекты-заглушки с заданными интерфейсами, у которых можно вызывать операции так, как это происходит в обычном объектно-ориентированном программировании.

Во время счета вместо объектов-заглушек в этот список системой OST подставляются ссылки на фактических соседей, т.е. на объекты, которые попали в окрестность данного экземпляра объекта.

Для иллюстрации приведем пример вызова функции *fun* у соседнего объекта под номером *i* в списке соседей:

```
# Вызов функции fun у соседа №i
self.neighbors[i].link.fun(<...>)
```

Для улучшения читаемости текста программы можно использовать синонимы для направлений соседства в конкретной топологии. Например, в случае двухмерной целочисленной решетки возможные соседи задаются однозначно: слева, справа, сверху и снизу. Приведем аналогичный пример вызова функции у соседнего объекта слева

```
# Вызов функции fun у соседа слева
self.left.fun(<...>)
```

Для иллюстрации использований топологий приведем примеры классов объектов прикладных систем.

```
# Пример класса объекта, иллюстрирующий работу со списком соседей.
class applied_object(ost.Object.Abstract):
    # В данном примере опущена техническая часть класса,
    # которая не относится к использованию соседей

    # Функция, которая входит в интерфейс.
    # Выводит на экран пришедшее сообщение
    def fun(self, message):
        print "Call function with", data

    # Функция проводящая вычисления
    def run(self):
        # Цикл по итерациям алгоритма
        while <...>:
            # перебираем всех соседей
            for neighbor in self.topology.neighbors:
                # вызываем у каждого соседа функцию fun
                neighbor.fun(<message>)



# Класс объекта с работой через синонимы соседей
class applied_object(ost.Object.Abstract):
    # В данном примере опущена техническая часть класса,
    # которая не относится к использованию соседей

    # Функция, которая входит в интерфейс.
    # Выводит на экран пришедшее сообщение
    def fun(self, message):
        print "Call function with", data

    # Функция проводящая вычисления
```

```
    def run(self):
        # Цикл по итерациям алгоритма
        while <...>:
            self.left.fun(<message>)
```

Для классов топологий, определенных пользователем, в системе OST предусмотрены механизмы задания своих синонимов.

3.1.4. Примеры стандартных топологий.

В данном разделе мы рассмотрим стандартные классы топологий, которые входят в систему OST.

Класс целочисленной решетки **ost.Topology.Mesh**.

Эта топология отличается простотой задания окрестности в пространстве любой размерности. Окрестность точки задается с помощью очень простого свойства — это множество точек, у которых ровно одна координата отличается на 1 от соответствующей координаты рассматриваемой точки. При использовании данного класса в конструкторе задается параметр размерности пространства. Для малых размерностей (1,2 и 3) предусмотрены синонимы «направлений соседства».

Пример использования:

```
# Задание топологии
<...>.topology = ost.Topology.Mesh(dimension = <размерность>)

# Доступные синонимы соседей в зависимости от размерности
# dimension = 1 — left, right
# dimension = 2 — left, right, up, down
# dimension = 3 — left, right, up, down, front, behind
```

Класс кольца **ost.Topology.Ring**.

Данная топология представляет собой граф в виде кольца, состоящий из N точек, пронумерованных от 1 до N. Окрестность точки состоит из соседей справа и слева. Топологию кольца удобно применять в задачах, требующих циклическую передачу данных между частями прикладной системы. Например, такая топология используется при параллельном умножении матриц, в котором полосы столбцов передаются через кольцо процессоров, в которых хранятся полосы строк. При использовании данного класса в конструкторе задается количество точек кольца.

Пример использования

```
# Задание топологии
<...>.topology = ost.Topology.Ring(N = <количество точек кольца>)

# Доступные синонимы соседей left, right
```

Класс неструктурированной решетки **ost.Topology.Graph**.

С помощью данной топологии задается произвольный граф. Окрестность вершины задается множеством вершин, с которыми соединена ребром данная. При использовании на практике пользователь может использовать различные интерфейсы конструктора.

Задание топологии с помощью таблицы, описывающей для каждой вершины ее окрестность

```
edges = { P1: [P1_neighbor_1,…,P1_neighbor_N1],
          …
          Pm: [Pm_neighbor_1,…,PM_neighbor_Nm]}
```

В данном примере `P1,…,Pm` — вершины графа, а список
`[Pi_neighbor_1,…,Pi_neighbor_Ni]`
задает набор вершин, входящих в окрестность i-ой точки.

```
# Задание топологии с помощью таблицы связности
<...>.topology = ost.Topology.Graph(edges = edges)
```

Задание топологии возможно так же с помощью сторонних генераторов графов, например, с помощью широко используемой системы Metis:

```
# Задание топологии с помощью Metis
<...>.topology = ost.Topology.Graph(metis_datafile = datafile.dat)
```

Класс фиксированной окрестности **ost.Topology.Neighborhood**.

С помощью данного класса топологии для любой точки пространства описывается фиксированная окрестность, состоящая из константного множества точек. В системе OST Класс фиксированной окрестности применятся в случае, когда требуется специализировать локальную топологию конкретного объекта.

В конструкторе задается список наборов координат точек, входящих в окрестность.

```
# Определение локальной топологии
object.topology = ost.Topology.Neighborhood([P1,…,Pk])
# Здесь Pi = [Pi_x1,…,Pi_xN] — набор координат i-го соседа
```

3.2. Разметка временем действий в прикладной системе

Предполагается, что каждый объект в рассматриваемой распределенной системе эволюционирует во время счета, проходя через последовательность состояний. Прикладному программисту предлагается помечать каждое состояние конкретным значением «локального времени объекта», которое хранится в специальной переменной.

Локальное время объекта может соответствовать физическому времени для моделируемого процесса внутри объекта или это может быть искусственное время, которое вводится лишь для упорядочения выполняемых действий. Например, это может быть порядковый номер итерации в алгоритме, заданном в объекте.

Синхронизация вычислений между объектами осуществляется системой OST на основе единственного правила – взаимодействие разрешается, если локальные времена объектов равны.

В процессе эволюции объекты выполняют запросы на продвижение своих локальных времен. Система OST обрабатывает эти запросы и в случае необходимости приостанавливает вычисления для соблюдения правила синхронизации.

Это правило в системе OST состоит в том, что для продвижения локального времени конкретного объекта до определенного значения необходимо наличие запросов в систему OST на продвижение локальных времен до этого же значения от всех соседей рассматриваемого объекта. Такое правило, во-первых, гарантирует исключение ситуаций с вызовом операции в объекте, у которого локального время больше времени вызывающего объекта. Во-вторых, обеспечивается достаточная свобода для проведения параллельного счета для несмежных объектов, так как допускается расхождение локальных времен между парой несмежных объектов на сумму приращений времен для всех промежуточных объектов, располагающихся на пути по связям между рассматриваемыми несмежными объектами.

3.2.1. Синхронизация вызовов между объектами

Взаимодействие одного объекта с другим объектом всегда происходит через служебный объект-ссылку. При неравенстве локальных времен вызывающего и вызываемого объекта система OST откладывает вызов до совпадения этих времен.

Объект-ссылка на «фактического» соседа в системе OST представляет собой специальный объект связи, который обеспечивает локальный вызов (в случае нахождения вызывающего и вызываемого объектов в одном адресном пространстве) или удаленный вызов (в случае нахождения вызывающего и вызываемого объектов на разных процессорах) с соблюдением сформулированного выше условия синхронизации.

3.2.2. Методы соблюдения условия синхронизации

В настоящее время известно много разных алгоритмов синхронизации по времени [4]. Один крайний случай – это ослабление условия разрешения вызова до требования, чтобы время вызывающего объекта было не меньше времени вызываемого. В этом случае разрешается много параллельных вычислений, но могут возникать ситуации, когда время в однажды вызванном объекте в результате этого вызова ушло вперед, а затем пришел вызов из другого объекта, локальное время которого оказалось меньше времени вызываемого объекта. В этом случае необходим откат системы к более раннему состоянию (возможно каскадный).

Другой крайний случай – это алгоритм, глобально упорядочивающий все локальные времена и допускающий вызов только в том случае, если он помечен глобально минимальным временем. В этом случае параллелизм счета будет отсутствовать. Вообще говоря, предполагается, что пользователю может быть предоставлен набор алгоритмов, из которого он сможет выбрать тот алгоритм, который является оптимальным для конкретной прикладной задачи.

По нашему мнению оптимальными для большинства случаев будут алгоритмы без откатов, которые являются промежуточными между описанными выше двумя крайностями. Правило синхронизации именно для такого алгоритма приведено в разделе 3.2.

3.3. Файл объектов

Для полноценной работы с системой OST пользователю нужны простые средства создания и хранения прикладных систем. Для решения этой задачи авторами работы предлагается использовать файл объектов — универсальный контейнер для хранения модели.

3.3.1. Создание модели

Для создания модели пользователь пишет программу создания модели. В ходе работы этой программы создается файл объектов, в который помещается множество объектов прикладной системы. Объекты, помещенные в файл, хранятся в «сериализованном» виде. Естественно, что в дальнейшем в момент счета прикладной задачи «сериализованные» объекты будут «подкачиваться» из файла в оперативную память, «десериализоваться» и «считаться». Объекты, составляющие модель, конечно, также могут создаваться и уничтожаться динамически в процессе основного счета.
После помещения в файл объектов всех объектов модели, в него помещаются все дополнительные компоненты, которые необходимы для функционирования прикладной системы. К таким компонентам относятся: данные (например, начальные и граничные условия), исходные коды, библиотеки и прочее.
В итоге работы программы получается единый файл объектов, который содержит в себе все необходимое для запуска и последующего функционирования модели на МВС.

“Почти” исполняемый пример, содержащий все этапы создания параллельной программной системы, приведен в Приложении 2.

3.3.2. Развертывание модели на МВС

В системе OST заложен принцип универсальности работы с файлами объектов при использовании на различных вычислительных мощностях. Для пользователя это означает, что модель можно развернуть на счет из файла объектов как на домашнем компьютере, так и на МВС, ничего специально не изменяя в самой модели или файле объектов.

В процессе подготовки модели к счету система OST в автоматическом режиме распределит объекты из файла объектов по доступным вычислительным мощностям и установит связи, согласно заданным топологиям. При необходимости для оптимизации использования вычислительных мощностей системой OST может быть задействован механизм подкачек, при использовании которого некоторые объектов могут быть перемещены между процессорами или вытесняться из счета в файл объектов.

Заметим, что при данной схеме работы естественным образом реализуется и механизм контрольных точек и рестартов. В файле объектов в любой момент времени хранится текущее состояние объектов модели и, возможно, несколько поколений этих состояний. Поэтому счет может быть возобновлен с того места, где он был по разным причинам остановлен, в том числе и на другой МВС.

4. Практические результаты

С помощью системы OST было запрограммированы и просчитаны на разных суперкомпьютерах как тестовые, так и реальные задачи из области газовой динамики.

В качестве одной из тестовых задач была использована задача параллельного умножения матриц, которая позволяет наглядно продемонстрировать ускорение расчетов при изменении количества параллельных частей. Для решения данной задачи был реализован алгоритм с ленточным разделением матриц [5]. В результате тестовых расчетов на МВС были получены результаты, демонстрирующие почти линейное ускорение при увеличении количества процессоров МВС (см. Приложение 1, Пример 1).

В качестве одной из реальных задач была реализована параллельная версия комплекса программ M2DGD для решения двухмерных задач газовой динамики [6]. Конкретно решалась задача об обтекании конического тела. Для этой задачи были проведены прикладные расчеты с использованием различного количества параллельных частей. Результаты работы параллельного комплекса программ полностью совпали с эталонными результатами, полученными в однопоточном расчете.

Было проведено сравнение эффективности распараллеливания по сравнению с параллельной версией M2DGD, реализованной с помощью MPI. На одинаковых параметрах задачи обе реализации давали практически идентичные времена счета. Это сравнение показало, что использование средств OST для автоматизации построения связей и синхронизации не привело к потере эффективности полученной вычислительной модели по сравнению с моделью, в которой те же проблемы решались вручную (см. Приложение 1, Пример 2).

По мнению авторов, результаты тестов и реальных расчетов продемонстрировали эффектность и простоту работы с системой OST.

5. Заключение

Основной результат работы, с точки зрения авторов, состоит в фактическом сведении сложности создания распределенных параллельных программных систем к сложности программирования локальных систем.

В предлагаемых новых программных средствах формализованы понятия "пространство-время". Такая формализация продолжает общую тенденцию развития средств программирования, заключающуюся в постепенной формализации общих понятий из прикладных областей. Примеры: сложные структуры данных предназначены для отображения на них сложных структур из прикладных областей; понятие процесса отражает понятие процесса изменения прикладной области; понятие объекта отражает прикладной объект и т.п. Формализация именно самых общих понятий позволяет эффективно вынести "за скобки" в операционную среду те действия, которые каждый прикладной программист вынужден был раньше повторять в каждой своей программе.

При создании параллельных программных моделей для любых физических областей при любом методе необходимо решить три основные проблемы:

1. Как формально описать параллельно эволюционирующие и взаимодействующие в процессе эволюции части физической области?

2. Как описать топологию связей между взаимодействующими частями параллельной системы? Для областей сложной структуры, отображаемых на десятки или сотни тысяч процессорных ядер, это сложная задача.

3. Как синхронизовать параллельный счет параллельно эволюционирующих программных (вычислительных) объектов?

Первая проблема относительно успешно уже решена, например, в объектно-ориентированных программных системах путем описания части физической области в виде программного объекта в смысле одного из О-О языков (С++, Ява, Питон). Решение двух других проблем в настоящее время практически полностью возлагается на прикладных программистов, которые раз за разом решают фактически одну и ту же задачу с помощью языковых средств низкого уровня.

Основная идея реализованного метода решения этих двух проблем заключается в формализации в виде соответствующих программных средств совершенно естественных для физических областей понятий "пространство-время".

Понятие "времени" давно используется в программировании, но в очень узких областях - моделировании дискретных устройств, моделировании систем управления в реальном времени. Как универсальный механизм синхронизации общего назначения, заменяющий известные в настоящий момент механизмы синхронизации параллельных действий, ранее не использовался.

Понятия "пространство" и "окрестность объекта" в виде "списка формальных соседей объекта" (принципиально обобщающее понятие - "список формальных параметров подпрограммы") ранее не использовались. Эти понятия позволяют упростить описание топологии связей, существенно используются при синхронизации путем введения "локального времени" в окрестности объекта. Физический смысл синхронизации по локальному времени заключается в том, что в случае конечной скорости распространения возмущений возможен параллельный счет «удаленных» друг от друга частей физической области для разных моментов времени, так как влияние просчитанных изменений распространяется только на некоторую окрестность объекта.

В заключение хотелось бы еще раз подчеркнуть, что проблемы задания связей между частями распределенной системы и синхронизации эволюции этих частей тем или иным способом должны решаться и решаются в любой системе для распределенных вычислений. Отличие данной работы в том, что решение этих проблем дается в общем виде на основе понятий, которые отражают только общие для всех рассматриваемых частных случаев свойства и не содержат таких свойств, которые присутствуют в одном частном случае, но отсутствуют в другом. Иногда высказывается мнение, что применение некоторого общего средства для широкого класса частных случаев будет во многих случаях неэффективно. Но это верно только тогда, когда делается попытка универсального использования средства, несущего специфику узкого класса случаев. Если же обобщение сделано корректно, то его конкретизация для частного случая в принципе не приводит к неэффективности.

6. Литература

[1] O'Hanlon, C. A conversation with John Hennessy and David Patterson. *Queue 4*, 10 (Dec. 2005/Jan. 2006), 14–22.

[2] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, Katherine Yelick, A view of the parallel computing landscape, Communications of the ACM Volume 52, Number 10 (2009), Pages 56-67

[3] Илюшин А.И., Колмаков А.А., Меньшов И.С., Построение параллельной вычислительной модели путем композиции вычислительных объектов,

[4] Causality Representation and Cancellation Mechanism in Time Warp Simulations
Malolan Chetlur and Philip A. Wilsey
Experimental Computing Laboratory
Dept. of ECECS, PO Box 210030, Cincinnati, OH 45221-0030

[5] http://math.csu.ru/~rusear/DipKurs/ParMetUmnMatr.html

[6] I. Menshov, Y. Nakamura, Hybrid Explicit-Implicit, Unconditionally Stable Scheme for Unsteady Compressible Flows, AIAA Journal, Vol. 42, No. 3, pp. 551-559, 2004.

[7] ost.kiam.ru

Приложение 1. Результаты счета двух примеров.

Расчеты производились на МВС rsc4.kiam.ru, состоящей из 64 узлов, на каждом из которых имеется по 2 процессора.

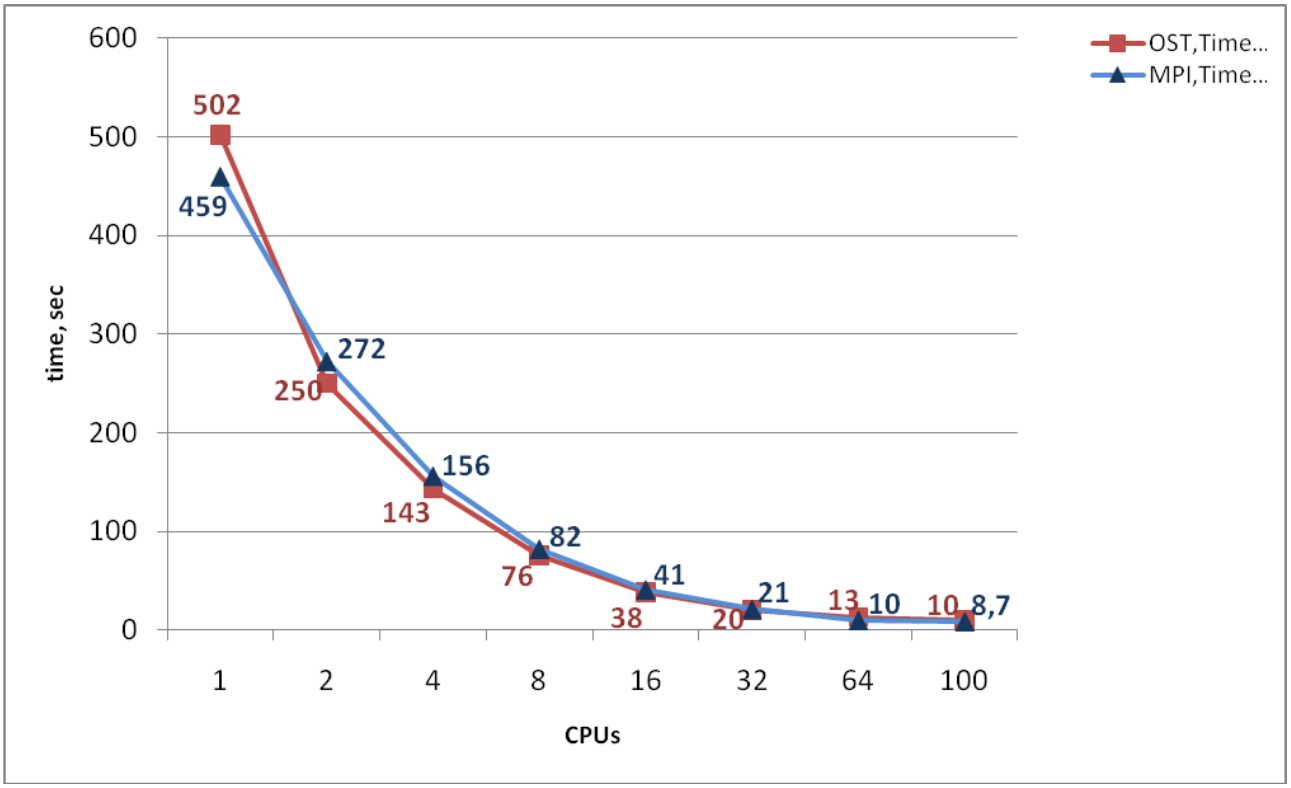
Пример 1. Умножение матриц.

В примере умножения матриц перемножались матрицы одинакового размера 1024x1024, заполненные случайными числами с плавающей точкой. Матрицы делились на различное число частей, в соответствие с алгоритмом ленточного деления матриц [5]. Результаты расчетов представлены в таблице:

Кол-во процессоров	**время**	**Эффективность**
1	1377,28 с	100 %
2	689,39 с	99 %
4	344,81 с	99 %
8	172,62 с	99 %
16	87,48 с	98 %
32	50,72 с	84 %
64	31,86 с	68 %

Пример 2. Расчет обтекания конуса.

В примере использования комплекса программ M2DGD проводилось сравнение 2-х различных параллельных реализаций: с автоматизацией построения связей и синхронизацией (OST), и с ручным построением связей и синхронизацией (MPI).



Приложение 2. Пример последовательности действий по созданию параллельной программной системы.

Рассмотрим последовательность шагов, которую должен выполнить прикладной программист для создания прикладной системы, предназначенной для запуска в среде OST.

Шаг 1. В первую очередь прикладным программистом выбирается классы топологии, которые будут использоваться в системе для задания связей между параллельно эволюционирующими частями. Выбор производится либо из реализованных в системе OST классов наиболее часто используемых топологий, либо описывается новый пользовательский класс топологии.

Приведем пример класса топологии в случае определения топологии с помощью функции описания окрестности.

```
# Класс с заданной функцией описания окрестности для топологии кольца
class applied_ring_topology(ost.Topology.Abstract):

    # Конструктор класса
```

```
    def __init__(self, N):
        self.N = N

    # Функция описания окрестности точки в пространстве
    def neighborhood(self, p):
        # self - аналог this в c++
        # p - набор(массив) координат, рассматриваемой точки

        # В переменную p_left поместим набор координат соседа слева
        if p[0] != 0: # Если p[0] не равен 0, то сосед на 1 меньше
            p_left = [p[0] - 1]
        else: # иначе замыкание через кольцо
            p_left = [N - 1]

        # В переменную p_right поместим набор координат соседа слева
        if p[0] != N - 1: # Если p[0] не равен N - 1, то сосед на 1 больше
            p_right = [p[0] + 1]
        else: # иначе замыкание через кольцо
            p_right = [0]

        # Возвратим описание окрестности
        # Синоним "left" соотвествует соседу слева
        # Синоним "right" соотвествует соседу справа
        return { "left": p_left,
                 "right": p_right
               }
```

Шаг 2. Следующий шаг при создании прикладной программы – это описание классов прикладных объектов, которые будут использоваться при создании частей распределенной прикладной системы. Стоит напомнить, что в описании такого класса используется понятие окрестности, задаваемое конкретным классом топологии.

```
# Общая структура класса прикладного объекта
class applied_object(ost.Object.Abstract):
    # Класс прикладного объекта

    # Класс содержит в себе функции, составляющие интерфейс объекта
    def fun_1(self, ...):
        # Содержимое функции
    # ...
    def fun_N(self, ...):
        # Содержимое функции

    # Функция, задающая интерфейс описываемого объекта с его окружением
    # в виде списка формальных соседей.
    # При вызове создает синонимы и интерфейсы для каждого соседа
    def init_topology(self):
        # Задаем интерфейс и синоним для формального 1-го соседа
        self.init_neighbor(i1, Class_interface_i1, "synonym_i1")
        # Задаем интерфейс и синоним для формального 2-го соседа
```

```
    self.init_neighbor(i2, Class_interface_i2, "synonym_i2")
    ...
    # Задаем интерфейс и синоним для формального K-го соседа
    self.init_neighbor(iK, Class_interface_iK, "synonym_iK")


# А так же функцию, которая запускает вычисления

# Функция проводящая вычисления
def run(self):
    # self - аналог this в c++
    # Обычно в run используется цикл по итерациям алгоритма
    for iteration in xrange(0, M):

        # Для обращения к соседям можно использовать список(массив)
        # self.topology.neighbors

        # Обращение к i-ому соседу по fun_j-ой функции
        self.topology.neighbors[i].link.fun_j()

        # При наличии синонимов использование проще
        # Обращение к соседу слева по fun_j-ой функции
        self.left.fun_j()

        # Для синхронизации используются продвижения по времени
        # В переменной self.time хранится текущее время объекта

        # Для продвижение на шаг time_step делается запрос к монитору OST
        # возврат из которого произойдет только после продвижения
        self.setXYZT(self.time + time_step)

        # Объекты могут менять свое положение в пространстве
        # Получение актуального массива координат текущего объекта
        coord = self.topology.get_coordinates()

        # Монитор OST изменяет координаты
        # только вместе с продвижением по времени
        self.setXYZT(self.time + time_step, coord)

    # После окончания вычислений завершаем вычисления
    self.setFinish()
```

Шаг 3. Последний шаг при создании прикладной системы – это создание программы инициализации. В данной программе создается файл объектов, в который прикладной программист помещает все необходимое для функционирования модели в параллельной среде. В программе поочередно создаются прикладные объекты на основе классов, описанных в шаге 2, которые связываются в единую модель с помощью классов топологий из шага 1. Помимо множества прикладных объектов, в файл объектов при необходимости помещаются файлы с исходными кодами, библиотеками и начальными данными.

```
#Пример программы инициализации

# Начинаем инициализацию модели
# Создаем объект инициализации.
# Сохранять будем в файл modelname.mod
obj_init = ost.Core.Init("modelname.mod")

# Зададим класс глобальной топологии
# В данном случае кольцо с 10 элементами
obj_init.topology = applied_ring_topology (N = 10)

# Дальше проводится цикл, в котором создаются объекты модели
# В данном примере создаются 10 объектов, для кольца
for index in xrange(0,10):
    # Создание объекта типа applied_object
    app_object = obj_init.create_object( applied_object )
    # Далее объект заполняется необходимыми данными

    # При необходимости можно задать локальную топологию окрестности объекта
    # В данном примере мы описываем окрестность, состоящую из элементов P1,…,Pk
    app_object.topology = ost.topology.Neighborhood([P1,…,Pk])

    # Помещаем объект в точку пространства объектов
    # Точка имеет координату index
    obj_init.topology.set(app_object, index)

# Добавление файла с программным кодом объекта
# С помощью аналогичных конструкций можно добавить и другие данные
obj_init.addSorceFile("applied_objects.py")

# Сохранение модели в файле
obj_init.save()
```

Итого. После запуска программы инициализации создается файл объектов, который может быть помещен в среду OST для параллельного выполнения на МВС.