Объектные модели высокого уровня для разработки эффективных программ с параллелизмом по данным*

О.И. Самоваров, С.С. Гайсарян, В.В. Бабкова

Аннотация. В данной статье рассматривается возможность создания параллельных программ средствами моделей параллелизма высокого уровня, которые отражают специфику прикладной задачи и учитывают особенности архитектуры конкретного вычислительного комплекса. Такие модели призваны скрыть от разработчика уровень MPI (Message Passing Interface) – среды передачи сообщений [1]. Использование объектных моделей параллелизма превращает процесс разработки параллельных программ заданной прикладной области в отработанную технологию. Это позволяет значительно сократить время, затрачиваемое на создание и отладку параллельных программ.

1. Введение

С более глубоким проникновением информационных технологий в жизнь человека увеличивается круг задач, для решения которых требуются значительные вычислительные ресурсы. Если раньше это были задачи, связанные с научно-техническими расчетами, то теперь вычислительные мощности, сопоставимые с суперкомпьютерными, требуются в самых разных областях человеческой деятельности. Например, моделирование графических трехмерных сцен в процессе дизайна самой разной направленности, организация оперативного поиска информации в хранилищах баз данных, обработка массированных запросов в компаниях, предоставляющих коммуникационные услуги.

В настоящее время, как альтернатива суперкомпьютерам появились и завоевывают все большую популярность вычислительные кластеры. Кластер – это система параллельной или распределенной обработки, которая состоит из узлов – отдельных компьютеров, соединенных между собой быстрой локальной сетью, например, Myrinet [2] и работающих совместно как один интегрированный вычислительный ресурс [3]. Узлом кластера может быть

* Работа поддерживалась грантом РФФИ 99-01-00206

одно- или многопроцессорный компьютер, имеющий память, устройство ввода-вывода и операционную систему. Конструктивно узлы кластера могут быть собраны в общем корпусе или разнесены в пространстве. Такие кластеры легко обслуживать, расширять и модернизировать. Они мобильны и относительно недороги, а их мощность сопоставима с суперкомпьютерной.

Суммируя сказанное, можно сделать вывод о том, что вопросы создания простых и эффективных средств разработки параллельных программ сегодня как никогда актуальны, что подтверждается и многочисленными исследованиями в этой области, результаты которых публикуются в научной периодике и на сайтах в Интернет [4]. Одним из таких средств является среда ParJava [5]. Среда ParJava расширяет среду Java [6] средствами разработки программ с параллелизмом по данным, как для однородных, так и для неоднородных параллельных вычислительных систем с распределенной памятью.

Программы для многопроцессорных вычислительных комплексов с распределенной памятью (суперкомпьютеров, кластеров, локальных сетей), как правило, создаются на языках C, C++ и Fortran, расширенных средствами, реализующими обмен сообщениями. В настоящее время наиболее широкое распространение получили два стандартных интерфейса, описывающих межпроцессорные взаимодействия, – MPI (Message Passing Interface) и PVM (Parallel Virtual Machine) [7]. Эти интерфейсы реализованы как библиотеки к языкам C, C++, Fortran почти для всех существующих параллельных платформ. Наибольшей популярностью у разработчиков пользуются реализации интерфейса MPI.

Параллельная программа, использующая MPI, может рассматриваться как система взаимодействующих процессов, каждый из которых выполняется на своем процессоре. От разработчика такой программы требуется распределить данные и вычисления, организовать межпроцессорные взаимодействия средствами обмена сообщений. Эффективность и масштабируемость параллельных MPI программ в большой степени может зависеть от архитектуры вычислительного комплекса, на котором она выполняется. И в некоторых случаях вместо ускорения выполнения программы можно получить обратный эффект. Поэтому процесс проектирования, реализации и отладки параллельной программы для некоторого вычислительного комплекса с распределенной памятью включает в себя не только разработку синтаксически и семантически корректного кода, но также и отладку эффективности его выполнения на этом вычислительном комплексе. Кроме того, отладка MPI программы затруднена еще и тем, что ее поведение в процессе выполнения недетерминировано. Несмотря на разнообразие функциональностей, MPI не предоставляет средств, необходимых для решения описанных выше задач, что делает разработку параллельных MPI программ трудоемким процессом.

В данной статье рассматривается возможность иного подхода к созданию параллельных программ. Этот подход основан на использовании моделей параллелизма, которые отражают специфику прикладной задачи и учитывают особенности архитектуры конкретного вычислительного комплекса. Такие модели призваны скрыть от разработчика уровень MPI -среды передачи сообщений, предоставив средства распараллеливания более высокого уровня [8]. Конечно, такой подход ограничивает круг решаемых задач в рамках одной модели, однако превращает процесс создания параллельных программ заданной прикладной области в отработанную технологию. Это позволяет значительно сократить время, затрачиваемое на разработку и отладку параллельных программ.

В качестве средства разработки моделей параллелизма рассматривается язык программирования Java. Таким образом, разрабатываемые модели будут обладать всеми преимуществами объектно-ориентированной технологии, а значит простыми в создании, модификации и отладке. Переносимость Java программ позволит использовать объектные модели параллелизма на разных вычислительных платформах. Кроме того, рассматриваемые объектные модели высокого уровня разрабатываются как часть среды ParJava.

Далее мы коротко рассмотрим возможности и состав объектной модели Java-DVM [8]. Также разберем два примера параллельных программ. Первая параллельная программа разработана с использованием объектной модели параллелизма Java-DVM, вторая - средствами языка Java, расширенного библиотекой MPI. Обе программы решают уравнение Лапласа методом Якоби и реализованы в рамках модели SPMD (Single Program Multiple Data) [3]. В заключении мы проведем сравнительный анализ этих примеров, что позволит нам яснее увидеть достоинства и недостатки использования объектных моделей параллелизма при разработке параллельных программ.

2. Объектная модель Java-DVM

Объектная модель Java-DVM представляет собой пакет Java классов, которые реализуют модель параллелизма DVM. Модель DVM была разработана в Институте прикладной математики им. М.В. Келдыша РАН [10]. Она предназначена для создания переносимых и эффективных вычислительных параллельных приложений. Аббревиатура DVM соответствует двум понятиям: Distributed Virtual Memory и Distributed Virtual Machine. Первое отражает наличие единого адресного пространства. Второе отражает использование виртуальных машин для двухступенчатой схемы отображения данных и вычислений на реальную параллельную машину.

В рамках модели разработчику предоставляются следующие возможности параллельного выполнения программы:

- выделение сети виртуальных процессоров, на которых будет выполняться параллельная программа. В дальнейшем эта сеть будет отображена на процессоры реального вычислительного комплекса;
- распределение данных;
- распределение вычислений;
- спецификация параллельно выполняющихся секций программы (параллельных задач) и отображение их на процессоры;
- организация эффективного доступа к удаленным (расположенным на других процессорах) данным;
- организация эффективного выполнения редукционных операций – глобальных операций с расположенными на различных процессорах данными, число которых уменьшается после выполнения (таких, как их суммирование или нахождение их максимального или минимального значения).

Кроме этого пользователю предлагаются средства отладки и анализа параллельной программы, о которых будет сказано ниже.

В рамках модели DVM предполагается, что параллельная программа разрабатывается для многопроцессорной системы с распределенной памятью, которую программист может представлять в виде 2-х архитектур – линейка или прямоугольная многомерная решетка процессоров. Это позволяет разработчику компактно и наглядно описывать распределение данных и вычислений. В объектной модели Java-DVM такое представление вычислительной среды описывается соответствующим классом пакета java.dvm – VPM (название класса от Virtual Parallel Machine). Создав экземпляр подобного класса, и изменяя его атрибуты, разработчик может управлять выделением сети виртуальных процессоров требуемой архитектуры, на которых будет выполняться его параллельная программа. В процессе запуска программы виртуальная сеть процессоров будет отображена на реальный вычислительный комплекс. При этом пользователю дается механизм управления этим отображением.

Распределение данных осуществляется разделением какого-либо из измерений определенного массива на примерно равные части и размещением этих частей на процессоры. Для примера на рисунке 1 показано y-распределение массива (вдоль столбцов) на линейку процессоров n1 – n4.

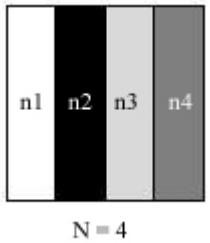

Рис. 1. Y-распределение массива на линейку процессоров.

Как было сказано выше, многопроцессорная вычислительная система, на которой будет выполняться параллельная программа, может рассматриваться не только как линейка процессоров, но и как многомерная решетка. Например, представим, что вычислительная система состоит из N процессоров. Тогда в рамках модели DVM она может быть представлена как 2-мерная матрица процессоров размерностью N1 на N2 (N=N1*N2). В этом случае одно из измерений массива может быть разделено на N1 отрезков, а какое-либо другое на N2 число отрезков – 2d-распределение. В результате массив будет разделен на N1*N2 частей, каждая из которых будет размещена на соответствующий процессор 2-мерной решетки. На рис. 2 показано распределение массива на 2-мерную решетку процессоров размерностью 2x2.

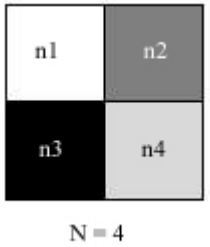

Рис. 2. 2d-распределение массива на двумерную решетку процессоров.

В определениях модели DVM распределяются только массивы, все остальные данные размещаются на процессорах целиком и называются размноженными. В Java-DVM предлагаются классы, описывающие x-, y- и 2d способы распределения массивов – `ColumnDistribution`, `RowDistribution` и `2dDistribution` соответственно. Определив способ распределения, разработчик может создать распределенный массив требуемого типа данных. Классы ряда `xxxDArray` описывают распределенные массивы для всех простых типов данных в языке Java. На рис. 3 показан пример создания распределенного массива в нотациях объектной модели Java-DVM.


```
import java.dvm.*;
public class MyClass
{
 public static void main (String[] args)
 {
  /* Выделяется сеть виртуальных процессоров. Сеть
  рассматривается как линейка процессоров. */
  VPM vpm = new VPM();

  /* Массив da размерностью 100x100 распределен на процессоры
  виртуальной сети vpm. Используется y-распределение */
  DoubleDArray da = new DoubleDArray(new RowDistribution(vpm,
  100, 100));
 }
}
```

Рис. 3. Пример реализации распределенного массива средствами библиотеки Java-DVM

Распределение вычислений определяется распределением данных. Разработчик параллельной программы может создать в своей программе параллельные циклы и указать для каждого из них «базовый» массив.

Базовым *массивом* называется массив, определяющий индексное пространство, по которому будет совершаться обход в параллельном цикле.

В этом случае витки этих циклов будут выполняться на тех процессорах, на которых размещены соответствующие элементы их базовых массивов. Предложения программы, расположенные вне параллельных циклов, выполняются на всех процессорах и, следовательно, вычисляют значения только размноженных переменных.

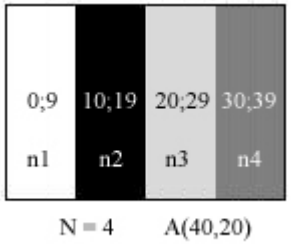

Рис. 4. Распределение витков параллельного цикла на линейку процессоров.

На рис. 4 показано распределение витков параллельного цикла на линейку процессоров n1-n4 совместно с матрицей *A* размерностью 40x20.

Для реализации параллельных циклов в Java-DVM есть классы блока `xxxDForall`. Это абстрактные классы, которые описывают параллельную обработку распределенных массивов для всех простых типов данных в языке Java. Разработчик должен создать свой собственный класс, расширяющий один из классов блока `xxxDForall`, в котором ему необходимо реализовать абстрактный метод `body()`. Этим методом описывается правило обхода элементов распределенного массива. На рис. 5 приведен пример, в котором показано, как реализуются параллельные циклы в Java-DVM.

```
import java.dvm.*;
public class MyClass
{
 public static void main (String[] args)
 {
  /* Выделяется сеть виртуальных процессоров. Сеть
  рассматривается как линейка процессоров. */
  VPM vpm = new VPM();
  /* Массив da размерностью 100x100 распределен на процессоры
  виртуальной сети vpm. Используется y-распределение */
  DoubleDArray da = new DoubleDArray(new RowDistribution(vpm,
  100, 100));
  /* Параллельный цикл с базовым распределенным массивом da
*/
  UserDCycle dc = new UserDCycle(da);

  /* Выполнить параллельный цикл */
  dc.execute();
 }
}

import java.dvm.*;
// Класс пользователя, в котором реализуется абстрактный
метод
//body()
class UserDCycle extends DoubleDForall
{
  DoubleDArray da; /* Базовый массив */
  public ACycleDJacobi(DoubleDArray a)
  {
    super(a);
    this.a = a;
  }
  /* Метод, в котором определяется порядок обхода
  элементов базового массива */
  public void body(int i, int j)
```



```
  {
    /* Записать в элемент (i,j) базового массива a значение
    суммы индексов i+j */
    a.putElem(i, j, i+j);
  }

}
```

Рис. 5. Пример реализации параллельного цикла средствами библиотеки Java-DVM.

Перед выполнением программы необходимо, чтобы все данные, которые требуются для вычислений, находились в локальной памяти узла. В случае параллельных циклов это условие выполняется редко. Очень часто случается ситуация, когда одна и та же ссылка на элемент массива в разных витках параллельного цикла вызывает обращение, как к локальным, так и нелокальным элементам массива. В DVM доступ к удаленным данным осуществляется путем их буферизации в памяти узлов вычислительной системы и обменом буферов сразу для нескольких витков цикла одновременно. Импортируемые элементы, которые примыкают к локальным (по их индикации в массиве) элементам, образуют так называемые грани. Грани имеют ширину. Например, предположим, что выделенная виртуальная сеть представляет собой линейку процессоров n1-n3 (рис. 6).

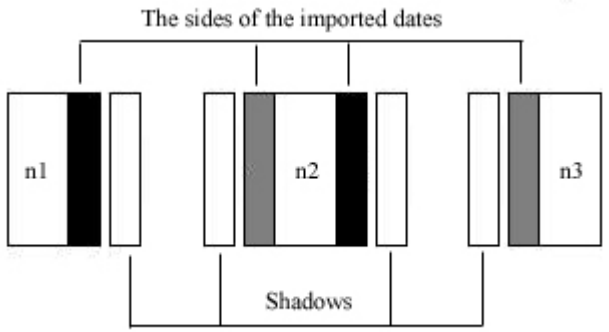

Рис. 6. Грани импортируемых данных и теневые грани.

Тогда грани импортируемых элементов располагаются на соседних процессорах, слева и справа. Увеличим память локальных секций распределенного массива на ширину граней импортируемых данных. Эта дополнительная память в модели DVM называется теневыми гранями секции

массива. Перед выполнением параллельного цикла грани импортируемых данных пересылаются на соседние процессоры и копируются в соответствующие теневые грани. Таким образом, осуществляется локализация удаленных данных. Выделение теневых граней, пересылка граней импортируемых данных, а также замена обращений к удаленным данным на обращения к их копиям в теневых гранях - все это осуществляется экземплярами классов реализующих распределенные массивы, т.е. блоком классов `xxxDArray`.

Модель DVM предоставляет разработчику средства отладки и анализа параллельных программ. Отладка параллельной DVM программы проходит в несколько этапов. На первом этапе программа отлаживается как последовательная. На втором этапе программа выполняется как параллельная в специальном режиме. В этом режиме промежуточные результаты параллельного выполнения сравниваются с эталонными результатами последовательного выполнения. Для осуществления этих этапов DVM предоставляет специальный инструмент – динамический отладчик.

Средства анализа производительности базируются на накоплении в памяти трассировки с временами выполнения определенных конструкций параллельной программы. Пользователь может получить информацию о некоторых характеристиках программы (или ее частей), позволяющих ему сделать заключения об эффективности ее выполнения.

Важно отметить, что разработчику предоставляются средства отладки в терминах самой модели.

3. Сравнение Java-DVM и Java+MPI

Теперь, когда мы познакомились с возможностями объектной модели параллелизма Java-DVM, рассмотрим два примера параллельной программы. Обе программы разработаны в рамках модели параллелизма по данным (SPMD) и решают уравнение Лапласа методом Якоби. Первая программа реализована средствами Java-DVM, вторая средствами языка Java расширенного библиотекой MPI.

Java-DVM. Параллельная программа представлена тремя классами: `DJacobi`, `DCycleA`, `DcycleB`, `InitDArray`. Класс `DJacobi` (рис. 7) реализует основную логику программы и содержит единственный метод. `main()`.


```
/* Решение уравнения Лапласа методом Якоби. Программа
разработана в рамках объектной модели параллелизма Java-DVM */
import java.dvm.*;
public class DJacobi
{
 // Определить количество итераций
 static final int MAXIT = 10;


 public static void main (String[] args)
 {
  // Получить размер матрицы
  //(Матрица квадратная размерностью ln x ln).
  int ln = Integer.valueOf(args[0]).intValue();


  /* Выделяется сеть виртуальных процессоров. Сеть
  рассматривается как линейка процессоров. */
  VPM vpm = new VPM();


  /* Массив da размерностью ln x ln распределен на процессоры
  виртуальной сети vpm. Используется y-распределение */
  DoubleDArray da = new DoubleDArray(new RowDistribution(vpm,
  ln, ln));


  /* Массив db размерностью ln x ln распределен на процессоры
  виртуальной сети vpm. Используется y-распределение */
  DoubleDArray db = new DoubleDArray(new RowDistribution(vpm,
  ln, ln));


  /* Инициализация распределенного массива в
  параллельном цикле */
  InitDArray ini = new InitDArray(db);
  /* Выполнить параллельный цикл инициализации */
  ini.execute();


  /* Определить параллельный цикл с базовым массивом ac */
  DCycleA ac = new DCycleA(da, db, 0, 0, da.getWidth(),
  da.getHeight(), 1, 1);
  /* Определить параллельный цикл с базовым массивом bc */
  DCycleB bc = new DCycleB(da, db, 1, 1, da.getWidth()-2,
  da.getHeight()-2, 1, 1);


  /* Итерационный цикл */
  for(int i=0; i<MAXIT; i++)
    {
```

```
        /* Выполнить паралельный цикл ac */
        ac.execute();
        /* Выполнить паралельный цикл bc */
        bc.execute();
        }
    }
}
```

Рис. 7. Решение уравнение Лапласа методом Якоби.

Классы DJacobi, DcycleA, InitDArray реализуют параллельные циклы.

Рассмотрим класс Djacobi подробнее. При решении задачи предполагается, что матрицы квадратные и имеют одинаковые размерности. Размер матриц определяется переменной ln, которой присваивается значение входного параметра.

```
int ln = Integer.valueOf(args[0]).intValue();
```

Следующее предложение программы создает объект, определяющий архитектуру сети виртуальных процессоров. Дальнее распределение данных и вычислений будет осуществляться именно на нее. В данном случае виртуальная сеть представляется как линейка процессоров.

```
VPM vpm = new VPM();
```

Далее создаются два распределенных массива. Эти массивы будут распределены методом y-распределения (вдоль столбцов массива) на виртуальную сеть процессоров vpm.

```
DoubleDArray da = new DoubleDArray(new
RowDistribution(vpm, ln, ln));
DoubleDArray db = new DoubleDArray(new
RowDistribution(vpm, ln, ln));
```

Этим предложением программы создается параллельный цикл инициализации распределенного массива. После того как цикл создан, его можно выполнить методом execute(). Вспомним, что InitDArray пользовательский класс, в котором реализуется абстрактный метод body() класса DoubleDForall. Именно в этом методе определяется предложение, обрабатывающее элементы массива. В данном случае элементы массива инициализируются предложением a.putElem(i, j, 1.0 + i + j). Подробнее об этом будет сказано ниже при рассмотрении классов, описывающих параллельные циклы.

```
InitDArray ini = new InitDArray(db);
ini.execute();
```


Создаются два параллельных цикла с базовым массивом da. Индексное пространство определяется входными параметрами конструктора классов DcycleA и DcycleB, а именно: 0, 0, da.getWidth(), da.getHeight() и 1, 1, da.getWidth()-2, da.getHeight()-2 соответственно. Методы getWidth(), getHeight() объекта da возвращают размер распределенной части массива. Фактически это означает, что параллельный цикл на каждом процессоре обработает только «свою» часть индексного пространства.

```
DCycleA ac = new DCycleA(da, db, 0, 0, da.getWidth(),
     da.getHeight(), 1, 1);
DCycleB bc = new DCycleB(da, db, 1, 1, da.getWidth()-2,
     da.getHeight()-2, 1, 1);
```

Далее в программе определяется итерационный цикл, в котором параллельные циклы выполняются MAXIT число раз.

```
for(int i=0; i<MAXIT; i++)
{
  ac.execute();
  bc.execute();
}
```

Теперь рассмотрим классы, описывающие параллельные циклы (рис. 8, 9, 10). Класс InitDArray описывает параллельный цикл инициализации распределенного массива. Класс DCycleA описывает параллельный цикл, в котором переопределяются значения элементов массива a значениями элементов массива b. Класс DCycleB описывает параллельный цикл, в котором в соответствии с семантикой задачи вычисляются значения элементов массива b.

```
/*
/ Параллельный цикл инициализации распределенного массива
*/
import java.dvm.*;
/* Пользовательский класс, описывающий параллельные циклы,
должен наследоваться от классов блока xxxDForall и реализовать
абстрактный метод body() */
class InitDArray extends DoubleDForall
{
  DoubleDArray a;

  /* Конструктор класса. Массив, полученный как параметр,
  определяется базовым */
  public InitDArray(DoubleDArray a)
  {
    super(a);
    this.a = a;
```

```
  }

  /* Реализуется абстрактный метод родительского класса
  DoubleDForall. В этом методе определяется предложение,
  обрабатывающее элементы распределенного массива */
  public void body(int i, int j)
  {
    a.putElem(i, j, 1.0 + i + j);
  }
}
```

Рис. 8. Параллельный цикл инициализации распределенного массива.

В классе `InitDArray` реализуется абстрактный метод родительского класса `DoubleDForall`. В этом методе определяется выражение, обрабатывающее распределенный массив. В данном случае это инициализация элементов распределенного массива *a* значениями выражения `a.putElem(i, j, 1.0 + i + j)`.

```
public void body(int i, int j)
{
  a.putElem(i, j, 1.0 + i + j);
}

/*
/ Класс DCycleA описывает параллельный цикл, в котором
/ переопределяются значения элементов массива a значениями
/ элементов массива b
*/
import java.dvm.*;


/* Пользовательский класс, описывающий параллельные циклы,
должен наследоваться от классов блока xxxDForall и
реализовать абстрактный метод body() */
class DCycleAi extends DoubleDForall
{
  DoubleDArray a, b;

  /* Конструктор класса. Массив a, полученный как параметр,
  определяется базовым */
  public ACycleDJacobi (DoubleDArray a, DoubleDArray b, int
  startx, int starty, int endx, int endy, int stepx, int
stepy)
  {
    super(a, startx, starty, endx, endy, stepx, stepy);
    this.a = a;
    this.b = b;
  }
```



```
  /* Реализуется абстрактный метод родительского класса
  DoubleDForall. В этом методе определяется предложение,
  обрабатывающее элементы распределенного массива */
  public void body(int i, int j)
  {
    a.putElem(i, j, b.getElem(i, j));
  }
}
```

Рис. 9. Параллельный цикл переопределения значений элементов массивов.

В классе `DCycleA` реализуется абстрактный метод родительского класса `DoubleDForall`. В этом методе определяется выражение, обрабатывающее распределенный массив. В данном случае в параллельном цикле переопределяются значения элементов массива *a* значениями элементов массива *b*.

```
public void body(int i, int j)
{
  a.putElem(i, j, b.getElem(i, j));
}

/*
/ Класс DCycleB описывает параллельный цикл, в котором в
/ соответствии с семантикой задачи вычисляются значения
элементов массива b.
*/
import java.dvm.*;

/* Пользовательский класс, описывающий параллельные циклы,
должен наследоваться от классов блока xxxDForall и реализовать
абстрактный метод body() */
class DCycleB extends DoubleDForall
{
  DoubleDArray a, b;

  /* Конструктор класса. Массив a, полученный как параметр,
  определяется базовым */
  public BCycleDJacobi(DoubleDArray a, DoubleDArray b, int
  startx, int starty, int endx, int endy, int stepx, int
  stepy)
  {
    super(b, startx, starty, endx, endy, stepx, stepy);
    this.a = a;
    this.b = b;
  }
```

```
  /* Реализуется абстрактный метод родительского класса
  DoubleDForall. В этом методе определяется предложение,
  обрабатывающее элементы распределенного массива */
  public void body(int i, int j)
  {
    b.putElem(i, j, 0.25* (a.getElem(i-1, j)+a.getElem(i+1,
j)+
    a.getElem(i, j-1)+a.getElem(i, j+1)));
  }
}
```

Рис. 10. Параллельный цикл, вычисляющий значения элементов массива.

В классе `DCycleB` реализуется абстрактный метод родительского класса `DoubleDForall`. В этом методе определяется выражение, обрабатывающее распределенный массив. В данном случае в параллельном цикле в соответствии с семантикой задачи вычисляются значения элементов массива *b*.

```
public void body(int i, int j)
{
  b.putElem(i, j, 0.25* (a.getElem(i-1, j)+a.getElem(i+1,
  j)+a.getElem(i, j-1)+a.getElem(i, j+1)));
}
```

Java + MPI. Рассмотрим теперь программу, решающую ту же задачу, но написанную на языке Java, расширенного библиотекой MPI. Класс `Jacobi` содержит единственный метод `main()`, в котором и реализована вся логика программы. Для большей наглядности разобьем программу на несколько блоков:

- инициализация (рис. 11);
- подготовка массива к распределению (рис. 12);
- распределение массива (рис. 13);
- локализация удаленных данных (рис. 14);
- итерационный цикл (рис. 15);
- завершение программы (рис. 16).

Рассмотрим каждый из них подробно.

Инициализация: В первых строках декларируются необходимые переменные. В качестве входного параметра ln программа получает значение размерности массива. Далее в этой части программы инициализируется библиотека MPI и определяются параметры виртуальной сети. Переменная myrank определяет текущий номер узла, на котором выполняется код программы. Это важный

параметр, поскольку дальнейшее поведение программы зависит именно от него.

```
/*
/  Решение уравнения Лапласа методом Якоби. Программа
разработана средствами языка Java, расширенного библиотекой
MPI
*/
import java.util.*;
import mpi.*;
class DJacobi
{
// Определить количество итераций
static final int MAXIT = 10;

// Массив
static float A[][] = new float[ln][ln];

// Размер рассылаемой части массива для каждого узла
int each;

/* Метод main */
public static void main(String args[])
{
  //  Получить  размер  матрицы.  (Матрица  квадратная
размерностью ln x ln).
  static int ln = Integer.valueOf(args[0]).intValue();

  // Буфер, необходимый для рассылки частей массива на
процессоры
  static float Ahelp[] = new float[ln*(ln - 1)];
  try
  {
  // Инициализация MPI
  MPI.Init(args);

  // Номер текущего узла
  int myrank = MPI.COMM_WORLD.Rank();


  // Размер сети
  int size = MPI.COMM_WORLD.Size();
```

Рис. 11. Инициализация.

Подготовка массива к распределению: В этом блоке происходит инициализация массива на нулевом узле. Прежде чем распределить двумерный массив, его надо представить в виде строки. У интерфейса MPI есть *ограничение* – методами MPI простые типы данных можно пересылать только

одномерными массивами. Далее вычисляются размеры рассылаемых частей массива и их смещение в буфере пересылки.

```
// Инициализация массива A
if(myrank == 0)
{
  for(int j = 0; j < ln; j++)
    for(int i = 0; i < ln; i++)
      A[i][j] = (float)(1.0 + i + j);
  int p = 0;
  // Подготовить буффер рассылки
  for(int j = 1; j < ln - 1; j++)
    for(int i = 0; i < ln; i++)
    {
      Ahelp[p] = A[i][j];
      p++;
    }
 }

// Размеры частей рассылаемого массива
int scounts[] = new int[size];
// Смешения этих частей в буфере рассылки
int disps[] = new int[size];

// Вычиление размеров частей массива, которые будут
рассылаться по процессорам
for(int i = 0; i < size; ++i)
  {
    scounts[i] = (int)Math.floor((float)ln / (float)size);
    if ( i < (ln - size*scounts[i]))
      scounts[i]++;
    each = scounts[myrank];
    scounts[i] *= ln;
    for(int j = 0; j < i; j++)
      disps[i] += scounts[j];
  }
```

Рис. 12. Подготовка массива к распределению.

Распределение массива: Массив распределяется равномерно по узлам сети методом y-распределения, - массив «режется» *вдоль* оси y. *Выделяется* память под локальные части массива на каждом процессоре. Далее части массива рассылаются методом Scatterv() по узлам виртуальной сети.

```
// Массивы, требующиеся для вычислений
float Blocal[][] = new float[ln][each];
float Alocalhelp[] = new float[ln*each];
```



```
  // Разослать части массива по узлам виртуальной сети
  MPI.COMM_WORLD.Scatterv(Ahelp,   0   ,   scounts,   disps,
MPI.FLOAT,
  Alocalhelp, 0, ln*each, MPI.FLOAT, 0);
  MPI.COMM_WORLD.Barrier();

  // Подготовить локальные массивы для вычислений
  int l = 0;
  for(int j = 1; j < each + 1; j++)
    for(int i = 0; i < ln; i++)
    {
      Alocal[i][j] = Alocalhelp[l];
      l++;
    }
```

Рис. 13. Распределение массив.

Локализация удаленных данных: Перед началом вычислений требуется локализовать удаленные данные. Для этого выделяется память под грани импортируемых данных SMas и *теневые* грани RMas. После этого каждый узел рассылает своим соседям импортируемые грани данных.

```
  // Грани импортируемых данных
  float[] SMas = new float [ln];

  // Теневые грани
  float[] RMas = new float [ln];

  // Разослать грани импортируемых данных и скопировать их в
  // теневые грани на левом узле
  if(myrank == 0)
  {
    for(int i = 0; i < ln; i++)
      Alocal[i][0] = A[i][0];
    for(int i = 0; i < ln; i++)
      SMas[i] = A[i][ln - 1];
    MPI.COMM_WORLD.Send(SMas,  0,  ln,  MPI.FLOAT,  size  -  1,
200);
  }
  // Разослать грани импортируемых данных и скопировать их в
  // теневые грани на  правом узле
  if(myrank == size - 1)
    {
      MPI.COMM_WORLD.Recv(RMas, 0, ln, MPI.FLOAT, 0, 200);
      for(int i = 0; i < ln; i++ )
        Alocal[i][each + 1] = RMas[i];
    }
  // Разослать грани импортируемых данных и скопировать их в
```

```
// теневые грани на всех других узлах
if(myrank > 0)
  {
    for(int i = 0; i < ln; i++)
      SMas[i] = Alocal[i][1];
    MPI.COMM_WORLD.Send(SMas, 0, ln, MPI.FLOAT, myrank - 1,
    200);
    MPI.COMM_WORLD.Recv(RMas, 0, ln, MPI.FLOAT, myrank - 1,
    200);
    for(int i = 0; i < ln; i++ )
      Alocal[i][0] = RMas[i];
  }
if(myrank < size - 1)
  {
    for(int i = 0; i < ln; i++)
      SMas[i] = Alocal[i][each];
    MPI.COMM_WORLD.Send(SMas, 0, ln, MPI.FLOAT, myrank + 1,
    200);
    MPI.COMM_WORLD.Recv(RMas, 0, ln, MPI.FLOAT, myrank + 1,
    200);
    for(int i = 0; i < ln; i++ )
      Alocal[i][each + 1] = RMas[i];
  }
```

Рис. 14. Локализация удаленных данных.

Итерационный цикл: В этом цикле осуществляются вычисления для каждой части массива. После каждой итерации *необходимо* произвести обновление значений теневых граней.

```
    /* Итерационный цикл */
    for(int k = 0; k < iternum; k++)//the main loop
    {
      for(int i = 1; i <= ln; i++)
        for(int j = 1; j <= each; j++)
          Blocal[i - 1][j - 1]=(float)0.25*(Alocal[i-1][j]+

Alocal[i+1][j]+Alocal[i][j-1]+Alocal[i][j+1]);

      // Обновление локального массива коэффициентов
      for(int i = 0; i < ln; i++)
        for(int j = 1; j < each + 1; j++)
          if((i != 0)&&(i != ln - 1))
            Alocal[i][j] = Blocal[i - 1][j - 1];

      //Обновление теневых граней.
      if(myrank > 0)    {
        for(int i = 0; i < ln; i++)
```



```
          SMas[i] = Blocal[i][0];
        MPI.COMM_WORLD.Send(SMas, 0, ln, MPI.FLOAT,
                            myrank - 1, 200);
        MPI.COMM_WORLD.Recv(RMas, 0, ln, MPI.FLOAT,
                            myrank - 1, 200);
        for(int i = 0; i < ln; i++ )
          Alocal[i + 1][0] = RMas[i];
      }
      if(myrank < size - 1)   {
        for(int i = 0; i < ln; i++)
          SMas[i] = Blocal[i][each - 1];
        MPI.COMM_WORLD.Send(SMas, 0, ln, MPI.FLOAT,
                            myrank + 1, 200);
        MPI.COMM_WORLD.Recv(RMas, 0, ln, MPI.FLOAT,
                            myrank + 1, 200);
        for(int i = 0; i < ln; i++ )
          Alocal[i + 1][each + 1] = RMas[i];
      }
    }
```

Рис. 15. Итерационный цикл.

Завершение программы: *Перед* выходом из программы необходимо вызвать метод `Finalize()`, который завершает работу библиотеки MPI.

```
      MPI.Finalize();
    }
    catch(MPIException e)
    {
      System.out.println(e.toString());
    }
  }
```

Рис. 16. Завершение программы.

Как видно из примеров, программа, разработанная средствами Java-DVM, имеет некоторые преимущества: Модель Java-DVM скрывает от разработчика параллельных программ уровень передачи сообщений, реализуя параллелизм программы классами своей библиотеки. Однако тут надо отметить, что методы классов библиотеки Java-DVM используют вызовы MPI для организации взаимодействия процессов. Используемая модель дает более наглядное представление о распределении данных и вычислений, поскольку архитектура виртуальной вычислительной среды и способы распределения данных задаются самим разработчиком. Программы, использующие библиотеку Java-DVM, легко модернизировать. Например, предположим, что в представленном выше примере программы необходимо изменить способ распределения массива - у-*распределение* на 2d-распределение. В случае программы на Java+MPI

пришлось бы добавить в код новые логические блоки, реализующие такой способ отображения массива на процессоры. Для этого в программе, разработанной средствами Java-DVM, для этого достаточно передать в качестве параметра конструктору, создающему распределенный массив, указатель на объект, реализующий 2d способ распределения, как показано ниже.

```
DoubleDArray da = new DoubleDArray(new 2dDistribution(vpm,
ln, ln));
```

Кроме того, DVM предоставляет средства отладки и анализа параллельных программ в определениях собственной модели.

Однако при рассмотрении модели Java-DVM как средства разработки параллельных программ необходимо учитывать следующее. Количество способов отображения данных на процессоры ограниченно. Кроме того, модель определяет структуры данных, которые могут быть распределены. Есть также ограничения на архитектуру вычислительной среды.

Можно сказать, что, с одной стороны, модель Java-DVM ограничивает своими определениями класс решаемых задач, с другой, позволяет разрабатывать эффективные параллельные программы в рамках определенной прикладной области.

4. Заключение

В данной статье была рассмотрена возможность использования объектных моделей высокого уровня для разработки эффективных программ с параллелизмом по данным. Была представлена объектная модель Java-DVM. Также проведено сравнение двух параллельных программ. Первая программа была разработана с использованием модели Java-DVM, вторая - средствами языка Java, расширенного библиотекой MPI .

Объектные модели параллелизма представляют собой технологию, способную обеспечить все этапы разработки эффективных параллельных программ: проектирование, реализацию и отладку.

В настоящее время ведется работа над объектной реализацией средств отладки параллельных Java-DVM программ.

Литература:

1. MPI: Message Passing Interface Standard, Message Passing Interface Forum, 2000; http://www.mcs.anl.gov/mpi/index.html
2. MPI: Message Passing Interface Standard, Message Passing Interface Forum, 2000; http://www.mcs.anl.gov/mpi/index.html
3. Rajkumar Buyya (ed.) "High Performance Cluster Computing": Programming and Applications. Vol. 2. Prentice Hall PTR, New Jersey, 1999.
4. MPI: Message Passing Interface Standard, Message Passing Interface Forum, 2000; http://www.mcs.anl.gov/mpi/index.html

5. А.И. Аветисян, И.В. Арапов, С.С. Гайсарян, В.А. Падарян. “Среда ParJava для разработки SPMD-программ для однородных и неоднородных сетей JavaVM.” Труды Института Системного Программирования Российской Академии Наук. Том 2, 2000.
6. JAVA 2 SDK, Standard Edition http://java.sun.com/products/jdk/1.2/
7. PVM: Parallel Virtual Machine, home page. http://www.epm.ornl.gov/pvm/pvm_home.html
8. С.С. Гайсарян, М.В. Домрачёв, В.Ф. Еч, О.И.Самоваров, А.И. Аветисян. “Параллельное программирование в среде Java для систем с распределенной памятью. Объектные модели параллельного выполнения.” Труды Института Системного Программирования Российской Академии Наук, Том 1, 1999.
9. В.П. Иванников, С.С. Гайсарян, М.В. Домрачев, О.И. Самоваров. “Объектная модель DVM и ее реализация на языке Java.” Вопросы Кибернетики. Приложения системного программирования. Выпуск 4. 1998
10. Н.А.Коновалов, В.А.Крюков, А.А.Погребцов, Ю.Л.Сазанов. “C-DVM - язык разработки мобильных параллельных программ”. Препринт ИПМ им. М.В.Келдыша РАН №86, 1997.