УДК 519.6

РЕАЛИЗАЦИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОЙ МОДЕЛИ МЕТОДА ДЕКОМПОЗИЦИИ НА ОСНОВЕ ПАРАЛЛЕЛЬНЫХ РАСПРЕДЕЛЕННЫХ КОМПОНЕНТОВ CORBA

С. П. Копысов[1], И. В. Красноперов[1], В. Н. Рычков[1]

Рассматриваются различные способы параллельной распределенной реализации объектной модели метода декомпозиции области с помощью технологий MPI и CORBA. Предлагается технология параллельных распределенных компонентов, основанная на компонентной модели CORBA, технологии асинхронного вызова методов AMI, инкапсуляции MPI-приложений.

1. Введение. Для решения сложных междисциплинарных задач требуется использовать высокопроизводительные аппаратно-программные среды. Большое количество разнородных архитектур ЭВМ и программных платформ приводит к необходимости создания единой технологии проектирования программного обеспечения (ПО), при этом важным параметром становится производительность вычислений. Среди существующих разработок особенный интерес вызывают те технологии, в основе которых лежат объектно-ориентированный и компонентный подход (ООП и КП), а также параллельные и распределенные вычисления. ООП и КП являются стандартами проектирования программного обеспечения; технологии параллельных распределенных вычислений позволяют создавать высокопроизводительные программы, абстрагируясь от платформы (промежуточное ПО).

Главным предметом данной работы является проектирование объектно-ориентированных вычислительных программ с помощью технологий MPI и CORBA, иллюстрация возможностей и недостатков этих технологий. Для этого будут рассмотрены различные способы реализации объектной модели метода декомпозиции области [1, 2], который достаточно легко поддается распараллеливанию. После анализа возможных решений, основанных на MPI и CORBA, предлагается новое решение — технология параллельных распределенных компонентов.

2. Вычислительная модель метода декомпозиции области. Для построения вычислительной модели используются методология ООП, которая позволяет строить абстрактные модели и описывает, каким образом они могут быть реализованы. Здесь и далее под моделью будем понимать программную объектно-ориентированную модель. Подробнее о проектировании ПО см. в [3, 4]. В вычислительной модели метода декомпозиции (МДО) можно выделить следующие модели:

— модель МДО; — модель распределенных данных в подобластях;

— модель параллельных вычислительных процессов в подобластях;

— модель эффективного распределения данных между подобластями (модель балансировки нагрузки).

Объектно-ориентированные модели метода конечных элементов и метода декомпозиции рассматриваются в [1,2].

2.1. Объектно-ориентированная модель МДО для МКЭ. Основные свойства МДО отражаются в системе объектов, которую удобно представить в следующем виде [1]:

1. Расчетная модель МКЭ описывает непрерывные тела и построенные для них конечно-элементные сетки, состоящие из элементов различного вида. Расчетная модель содержит следующие объекты: `Solid`, `Domain`, `Element`, `Node`. Для реализации МДО вводится ряд новых компонентов: `PartitionedDomain`, `SubDomain`, `DomainPartitioner`, `Graph`, `GraphPartitioner`. Базовые компоненты для разбиения графов — `Graph` и `GraphPartitioner`. Класс `DomainPartitioner` отвечает за разбиение области `PartitionedDomain`, включающей граф, листьями которого являются элементы, а остальными узлами — подобласти, объекты типа `SubDomain`.

2. Вычислительная модель МКЭ сопоставляет сеткам численные данные (системы линейных уравнений, матрицы и вектора). Для этого путем наследования от соответствующих объектов геометрической модели переопределяются сеточные объекты: `Domain`, `Element`, `Node`. В этих объектах появляются данные типа `LinearSOE` (Linear System Of Equations), `Matrix`, `Vector`. Реализация МДО требует

[1] Институт прикладной механики Уральского отделения РАН, ул. Горького, 222, 426000, г. Ижевск; e-mail: kopyssov@udman.ru

дополнить описание объекта `SubDomain` геометрической модели данными, соответствующими подобласти. Кроме этого, вычислительная модель МКЭ подробно описывает основные этапы процесса вычислений. Для этого в вычислительной модели определяются объекты `Analysis`, `Algorithm`, `Solver`. Вычислительная модель МДО расширяет модель МКЭ такими объектами, `DomainDecompositionAnalysis`, `DomainDecompositionAlgorithm`, `DomainSolver`. Контейнер объектов, необходимых для решения задачи МДО, — `DomainDecompositionAnalysis`; `DomainDecompositionAlgorithm` определяет общий процесс вычисления; `DomainSolver` решает систему уравнений, соответствующих одной подобласти.

2.2. Распределенные данные. Это набор однотипных объектов, соответствующих тому или иному сегменту данных декомпозированной задачи. Для МДО распределение данных необходимо не только для эффективной параллельной обработки, но еще и потому, что в реальных больших задачах хранение всех данных на одном вычислительном узле может оказаться невозможным. В алгоритме МДО распределенными данными являются конечно-элементная сетка и система уравнений.

2.3. Параллельные процессы. Это набор однотипных объектов, выполняющих те или иные методы одновременно, в общем контексте. Для МДО распараллеливание является достаточно легко формализуемым способом повышения производительности вычислений. В алгоритме МДО распараллеливанию подлежит, например, формирование и решение систем линейных уравнений на подобластях.

2.4. Балансировка нагрузки. МДО предполагает балансировку нагрузки на уровне прикладной системы. Она связана с эффективным распределением данных между подобластями таким образом, чтобы вычислительные затраты на каждом узле были примерно равны. В расчетной модели МДО были введены объекты `PartitionedDomain` и `DomainPartitioner`, описывающие граф подобластей (`Graph`) и метод его разбиения (`GraphPartitioner`). Таким образом, для эффективного распределения данных необходимо разработать систему объектов, унаследованных от `GraphPartitioner`, которые включают различные методы разбиения графа, а также подобрать для конкретной задачи наиболее приемлемый алгоритм разбиения.

Обзор различных алгоритмов перестроения сеток приведен в [5]. В настоящее время достаточно широко используется пакет Metis (а также его параллельная MPI-версия ParMetis) включающий богатый набор алгоритмов разбиения графов. Входными и выходными данными обоих пакетов являются массивы, которые в совокупности описывают граф.

3. MPI реализация метода декомпозиции. Технология MPI (Message Passing Interface — интерфейс обмена сообщениями) – стандарт и промежуточное ПО для организации параллельных вычислений. Основные описания — процессы; основные операции — различные типы обменов данными между процессами. Процессы выполняют единую инструкцию над своими данными и обмениваются полученными результатами. При необходимости дифференцировать процессы друг от друга в программу вводятся участки кода, соответствующие различным альтернативным вариантам, и в режиме выполнения в каждом из процессов, в зависимости от идентификатора процесса, запускается тот или иной код. Таким образом, MPI-программа представляет собой единый код для всех процессов, который разветвляется в тех местах, где процессы отличаются (рис. 1). Эту технологию целесообразно применять для реализации SIMD-моделей, когда все процессы в программе идентичны и структуры данных достаточно просты. Более подробную информацию о технологии MPI, ее развитии и соответствующие ссылки можно найти в [6].

3.1. Реализация объектов в MPI. Стандарт MPI не имеет средств описания объектов, но, как правило, реализованное промежуточное ПО MPI предназначено для разработки на языке C; следовательно, это ПО может быть использовано в рамках языка C++. Таким образом, C++ становится средством реализации объектов в MPI-программах. Традиционные объекты C++ используются в тех участках кода, которые выполняются в рамках одного MPI-процесса.

Рассмотрим способы реализации объектов-данных и объектов-процессов, которые используются в рамках всей системы MPI.

3.2. Распределенные данные. Распределенные данные можно реализовать двумя способами: перемещаемые объекты и распределенные объекты. Первые представляют собой обыкновенные объекты C++, включающие дополнительно методы по передаче и получению своего состояния с одного MPI-процесса на другой. Такой подход целесообразно использовать для перемещения объектов. При реализации удаленных объектов возникает проблема синхронизации их удаленного состояния с состояниями, хранящимися в других процессах. Один из вариантов перемещаемых объектов представлен в системе TPO++ [6].

Для примера рассмотрим перемещение объекта `NodalLoad`, представляющего узловую нагрузку. Для начала введем систему вспомогательных объектов, которая является базой при реализации перемещения объектов. Класс "идентифицируемый объект" `TaggedObject` содержит уникальный номер `tag`, определя-

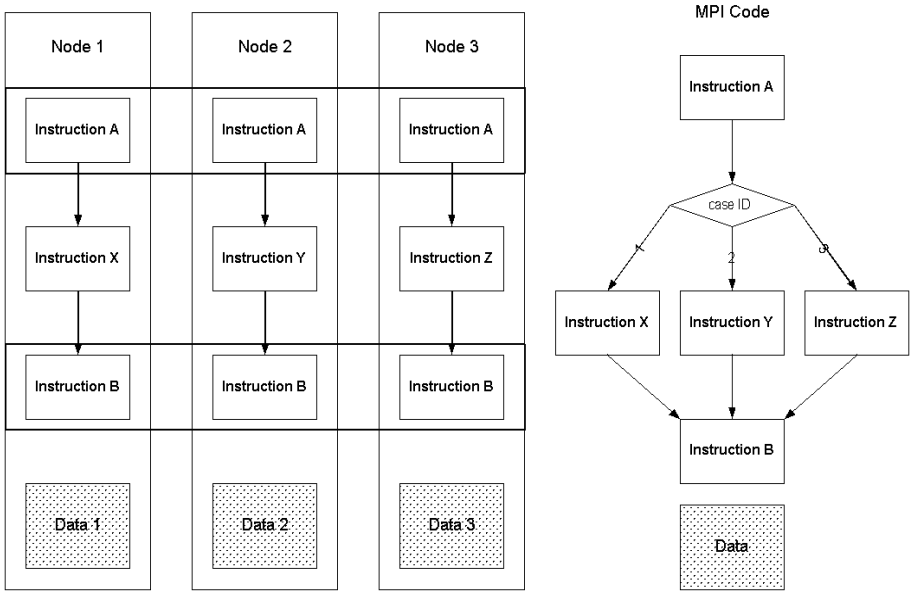

Рис. 1. MPI-программа

ющий экземпляр данного класса.

```
class TaggedObject {
 public:
   TaggedObject(int tag);
   virtual ~TaggedObject();
   inline int getTag(void) const;

 protected:
   void setTag(int newTag);

 private:
   int theTag;
};
```

Класс `Channel` — шина обмена элементарными данными (потоками байтов, идентификаторами, векторами/матрицами, объектами) между вычислительными узлами. Реализация класса `MPI_Channel` основана на обменах данными между MPI-процессами. Класс `MovableObject` — базовый класс для перемещаемых объектов. Его основными методами являются `sendSelf/recvSelf`: отправить и получить свое состояние в виде потока байтов через канал обмена данными. При отправке и получении состояния важную роль играет идентификатор, благодаря которому точно определяется адресат. Далее введем прикладные классы: абстрактный класс для объектов области `DomainComponent`, абстрактный класс нагрузки `Load` и, наконец, класс узловой нагрузки `NodalLoad`. В состояние объектов узловой нагрузки, кроме идентификатора, входит тип и вектор нагрузки.

В основе реализации распределенных объектов лежит процедура маршалинга — удаленный вызов методов объекта, передача параметров и прием результатов (RPC/CORBA). Распределенные объекты обеспечивают доступ к данным, расположенным в адресном пространстве другого MPI-процесса, позволяя избежать пересылки всего содержимого объекта. Такой подход является более универсальным и используется как для перемещаемых объектов, так и для объектов, имеющих стабильное место расположения. Различные способы реализации маршалинга представлены в системах TPO++, Charm++, PAWS [6], созданных на основе MPI.

```
class Channel {
 public:
   Channel ();
   virtual ~Channel();
   virtual int sendObj(int commitTag, MovableObject &theObject, ChannelAddress *theAddress =0) =0;
   virtual int recvObj(int commitTag, MovableObject &theObject, FEM_ObjectBroker &theBroker,
```

```
                ChannelAddress *theAddress =0) =0;
    virtual int sendID(int dbTag, int commitTag, const ID &theID, ChannelAddress *theAddress =0) =0;
    virtual int recvID(int dbTag, int commitTag, ID &theID, ChannelAddress *theAddress =0) =0;
    ...
};

class MPI_Channel : public Channel { };

int MPI_Channel::sendObj(int commitTag, MovableObject &theObject, ChannelAddress *theAddress) {
    return theObject.sendSelf(commitTag, *this);
}

int MPI_Channel::recvObj(int commitTag,MovableObject &theObject, FEM_ObjectBroker &theBroker,
    ChannelAddress *theAddress) {return theObject.recvSelf(commitTag, *this, theBroker);
}

int MPI_Channel::recvID(int dbTag, int commitTag, ID &theID, ChannelAddress *theAddress) {
    int nleft,nread;
    int *data = theID.data;
    char *gMsg = (char *)data;;
    nleft =  theID.sz;
    MPI_Status status;
    MPI_Recv((void *)gMsg, nleft, MPI_INT, otherTag, 0, otherComm, &status);
    int count =0;
    MPI_Get_count(&status, MPI_INT, &count);
    if (count != nleft) {
      cerr << "MPI_Channel::recvID() -";
      cerr << " incorrect number of entries for ID received ";
      return -1;
    }
    else
      return 0;
}

int MPI_Channel::sendID(int dbTag, int commitTag, const ID &theID, ChannelAddress *theAddress) {
    int nwrite, nleft;
    int *data = theID.data;
    char *gMsg = (char *)data;
    nleft =  theID.sz;
    MPI_Send((void *)gMsg, nleft, MPI_INT, otherTag, 0, otherComm);
    return 0;
}

class MovableObject {
  public:
    MovableObject(int classTag, int dbTag);
    MovableObject(int classTag);
    virtual ~MovableObject();
    int getClassTag(void) const;
    int getDbTag(void) const;
    void setDbTag(int dbTag);
    virtual int sendSelf(int commitTag, Channel &theChannel) =0;
    virtual int recvSelf(int commitTag, Channel &theChannel, FEM_ObjectBroker &theBroker) =0;
  private:
    int classTag;
    int dbTag;
}

class DomainComponent: public TaggedObject, public MovableObject;

class Load : public DomainComponent;

class NodalLoad : public Load
{
  public:
    NodalLoad(int classTag);
    NodalLoad(int tag, int node, int classTag);
    NodalLoad(int tag, int node, const Vector &load, bool isLoadConstant = false);
    ~NodalLoad();
    virtual void setDomain(Domain *newDomain);  virtual int getNodeTag(void) const;
    virtual void applyLoad(double loadFactor);
    virtual int sendSelf(int commitTag, Channel &theChannel);
```

```
    virtual int recvSelf(int commitTag, Channel &theChannel,FEM_ObjectBroker &theBroker);
  private:
    int  myNode;
    Node *myNodePtr;
    Vector *load;
    bool  konstant;
};

int NodalLoad::sendSelf(int cTag, Channel &theChannel) {
    ID data(5);
    data(0) = this->getTag();
    data(1) = myNode;
    if (load != 0)
    data(2) = load->Size();
    else
    data(2) = 0;
    data(3) = konstant;
    data(4) = this->getLoadPatternTag();

    int result = theChannel.sendID(dataTag, cTag, data);
    if (result < 0) {
    cerr << "NodalLoad::sendSelf - failed to send data\n";
    return result;
    }
    if (load != 0){
    int result = theChannel.sendVector(dataTag, cTag, *load);
    if (result < 0) {
        cerr << "NodalLoad::sendSelf - failed to Load data\n";
        return result;
    }
    }
    return 0;
}

int NodalLoad::recvSelf(int cTag, Channel &theChannel, FEM_ObjectBroker &theBroker) {
    int result;
    int dataTag = this->getDbTag();
    ID data(5);
    result = theChannel.recvID(dataTag, cTag, data);
    if (result < 0) {
      cerr << "NodalLoad::recvSelf() - failed to recv data\n";
      return result;
    }
    this->setTag(data(0));
    myNode = data(1);
    int loadSize = data(2);
    konstant = data(3);
    this->setLoadPatternTag(data(4));
    if (loadSize != 0) {
    load = new Vector(data(2));
    result = theChannel.recvVector(dataTag, cTag, *load);
    if (result < 0) {
      cerr << "NodalLoad::recvSelf() - failed to recv load\n";
      return result;
    }
    }
    return 0;
}
```

Рассмотрим код, реализующий процедуру маршалинга. Основными объектами являются `Actor` и `Shadow`. Первый выполняет методы на стороне сервера, второй является заместителем удаленного объекта на клиентской стороне.

```
class Actor {
  public:
    Actor(Channel &theChannel, FEM_ObjectBroker &theBroker, int numActorMethods);
    virtual ~Actor();
    virtual int  addMethod(int tag, int (*fp)());
    virtual int  getMethod();
    virtual int  processMethod(int tag);
    Channel         *getChannelPtr(void) const;
```

```
    FEM_ObjectBroker    *getObjectBrokerPtr(void) const;
    ChannelAddress      *getShadowsAddressPtr(void) const;
  protected:
    FEM_ObjectBroker *theBroker;
    Channel *theChannel;
  private:
    int numMethods, maxNumMethods;
    ActorMethod **actorMethods;
    ChannelAddress *theRemoteShadowsAddress;
};

class Shadow {
  public:
    Shadow(Channel &theChannel, FEM_ObjectBroker &theBroker, ChannelAddress &theAddress);
    Shadow(char *program, Channel &theChannel, FEM_ObjectBroker &theBroker,
       MachineBroker &theMachineBroker, int compDemand, bool startShadow);
    virtual ~Shadow();
    Channel           *getChannelPtr(void) const;
    FEM_ObjectBroker    *getObjectBrokerPtr(void) const;
    ChannelAddress    *getActorAddressPtr(void) const;
  protected:
    FEM_ObjectBroker      *theBroker;
    Channel       *theChannel;
  private:
    ChannelAddress    *theRemoteActorsAddress;
};
```

В приложении данной конструкции к методу декомпозиции рассмотрим следующие прикладные объекты: базовый класс `Subdomain`, определяющий общую функциональность подобласти, и распределенную версию `ActorSubdomain` — `ShadowSubdomain`.

```
class Subdomain: public Element, public Domain {
  public:
    Subdomain(int tag);
    Subdomain(int tag,
          TaggedObjectStorage &theInternalNodeStorage, TaggedObjectStorage &theExternalNodeStorage,
          TaggedObjectStorage &theElementsStorage, TaggedObjectStorage &theLoadPatternsStorage,
          TaggedObjectStorage &theMPsStorage, TaggedObjectStorage &theSPsStorage);

    virtual  ~Subdomain();

    virtual int buildSubdomain(int numSubdomains, PartitionedModelBuilder &theBuilder);
    ...
};
```

При запуске (метод `run`) сервер `ActorSubdomain` ожидает MPI-пакеты, разбирает их и выполняет соответствующую операцию, например формирование подобласти `buildSubdomain`.

```
class ActorSubdomain: public Subdomain, public Actor {
  public:
    ActorSubdomain(Channel &theChannel, FEM_ObjectBroker &theBroker);
    virtual ~ActorSubdomain();
    virtual int run(void);
    ...
};
int ActorSubdomain::run(void) {
    Vector theVect(2);
    bool exitYet = false;
    while (exitYet == false) {
    this->recvID(msgData);
    int action = msgData(0);

    int theType, tag, dbTag, loadPatternTag;
    Element *theEle;
    Node *theNod;
    SP_Constraint *theSP;
    MP_Constraint *theMP;
    LoadPattern *theLoadPattern;
    NodalLoad *theNodalLoad;
    ElementalLoad *theElementalLoad;
```

```
    DomainDecompositionAnalysis *theDDAnalysis;
    const Matrix *theMatrix;
    const Vector *theVector;
    PartitionedModelBuilder *theBuilder;
    const ID *theID;

    switch (action) {
      case ShadowActorSubdomain_buildSubdomain:
        theType = msgData(1);
        tag = msgData(3); // subdomain tag
        this->setTag(tag);
        tag = msgData(2); // numSubdomains
        theBuilder = theBroker->getPtrNewPartitionedModelBuilder(*this,
                                    theType);
        this->recvObject(*theBuilder);
        this->buildSubdomain(tag, *theBuilder);

        break;
        ...
    }
    }
    return 0;
}
```

Заместитель объекта "подобласть" на клиентской стороне формирует MPI-пакеты соответствующей операции, например `buildSubdomain`.

Для реализации сложных структур данных, которые необходимо распределять между вычислительными процессами, технологии MPI не хватает подхода распределенных объектов: не реализована процедура маршалинга.

3.3. Параллельные процессы. Параллельные процессы — объекты, инкапсулирующие свойства MPI-процессов, которые описываются в интерфейсе MPI. Для реализации этих классов удобно использовать объектно-ориентированные интерфейсы MPI, такие как MPI++, OOMPI [6], и подход распределенных объектов.

Параллельные процессы могут быть реализованы на базе конструкции `Actor/Shadow`, описанной выше. Рассмотрим код объекта `IncrementalIntegrator` по формированию матрицы жесткости для подобласти. Формирование матриц происходит параллельно в подобластях, результаты добавляются в систему линейных уравнений по мере завершения создания матриц.

Реализовать объекты-процессы в MPI достаточно просто, поскольку основным описаниями интерфейса MPI являются параллельные процессы и способы их взаимодействия. Проблема динамического запуска новых процессов решена в MPI-2.

```
class ShadowSubdomain: public Shadow, public Subdomain {
  public:
    ShadowSubdomain(int tag, Channel &theChannel, MachineBroker &theMachineBroker,
            FEM_ObjectBroker &theObjectBroker, int compDemand =0, bool startShadow = true);

    virtual  ~ShadowSubdomain();
    virtual int buildSubdomain(int numSubdomains, PartitionedModelBuilder &theBuilder);
    ...
};

int ShadowSubdomain::buildSubdomain(int numSubdomains,
PartitionedModelBuilder &theBuilder) {
  buildRemote = true;
  gotRemoteData = false;

  msgData(0) = ShadowActorSubdomain_buildSubdomain;
  msgData(1) = theBuilder.getClassTag();
  msgData(2) = numSubdomains;
  msgData(3) = this->getTag();
  this->sendID(msgData);

  this->sendObject(theBuilder);

  this->domainChange();
  return 0;
}
```

```
IncrementalIntegrator::formTangent
{
        FE_EleIter theEles = theAnalysisModel->getFEs();
        theLinearSOE->zeroA();
        while ((feElePtr = theEles()) != 0)
              feElePtr->formTangent(theIntegrator);
        while ((feElePtr = theEles()) != 0)
              theLinearSOE->addA(feElePtr->getTangent(), feElePtr->getID());
}
```

3.4. Использование прикладных MPI библиотек. Балансировку нагрузки в MPI версии МДО можно реализовать на основе параллельной библиотеки `ParMetis`. Для этого необходимо путем наследования от `GraphPartitioner` создать распределенный объект (`Actor/Shadow`) `ParMetisPartitioner` и перегрузить функцию `Partition`, которая переводит объектную информацию о графе подобластей во входные массивы библиотеки `ParMetis`, выполняет MPI-вызов той или иной функции разбиения графа и переводит выходные массивы в объекты. Объекты `ParMetisPartitioner` запускаются в нескольких экземплярах и выполняются параллельно. Кроме этого, библиотека ParMetis включает функции динамической балансировки, которые также доступны в параллельном распределенном объекте-разделителе.

Технология MPI процедурно ориентирована, в ней отсутствуют возможности описания объектов. В соответствии с новыми требованиями к проектированию ПО, в MPI необходимо ввести понятия параллельных распределенных объектов, что демонстрируют новые технологии и новое промежуточное ПО на базе MPI.

4. CORBA реализация метода декомпозиции. CORBA (Common Object Request Broker Architecture — общая архитектура брокеров объектных запросов) — стандарт и промежуточное ПО для работы с распределенными объектами. Основные описания — объекты; основные операции — различные типы вызовов методов объектов на основе маршалинга. Объекты выполняют операции, определяемые своими методами, при этом обращаются к своим данным. Методы могут вызываться синхронно и асинхронно, выполняться последовательно и параллельно. Технологию CORBA целесообразно использовать для реализации программ, в которых операции разнородны, а структуры данных достаточно сложны. Многопоточные CORBA-объекты реализуют модель вычислений MISD. Подробнее о технологии CORBA, а также сравнение ее возможностей с MPI можно найти в [6].

4.1. Реализация объектов в CORBA. Технология CORBA — промежуточное ПО для объектно-ориентированной разработки программ на различных языках программирования, как правило, объектно-ориентированных (C++, Java и др.). Независимость от языков программирования достигается путем предварительного описания объектов на специальном языке IDL (Interface Definition Language — язык описания интерфейсов) и компиляции этих описаний (интерфейсов) в код на языке программирования, на котором впоследствии объекты должны быть реализованы. При разработке CORBA-объектов, кроме традиционных объектов языка программирования, возникают объекты специального вида (рис. 2):

— сервант: прикладной объект,

— стаб и скелетон: объекты на стороне клиентского и серверного процессов соответственно, обеспечивающие удаленное обращение к серванту.

Код скелетона `POA_Domain::IDomain` и стаба `Domain::IDomain` автоматически генерируется по интерфейсу `IDomain`. Эти классы не требуют какого-либо редактирования со стороны разработчика и реализуют различные способы взаимодействия с объектом “область” (синхронные и асинхронные). Далее необходимо реализовать непосредственно сам прикладной объект (сервант) область `IDomain_I`, создать код всех его атрибутов и методов, которые могут быть вызваны удаленно. Ниже приведен фрагмент интерфейса CORBA объекта область `IDomain`.

Тройка специализированных объектов и составляет объект CORBA, который может быть запущен в одном узле или процессе ВС, а использован в другом. Так же как в объектно-ориентированном программировании, на базе одного виртуального интерфейса можно реализовать несколько разных объектов, на базе одного IDL-интерфейса/стаба/скелетона можно реализовать несколько разных CORBA-объектов. Все они будут иметь одинаковое описание, но разное поведение.

```
module Domain {
    interface IDomain {
        readonly attribute long NumberOfElements;
        ...
        Domain::IElement ElementItem (in long tag);
        ...
    };
    ...
```

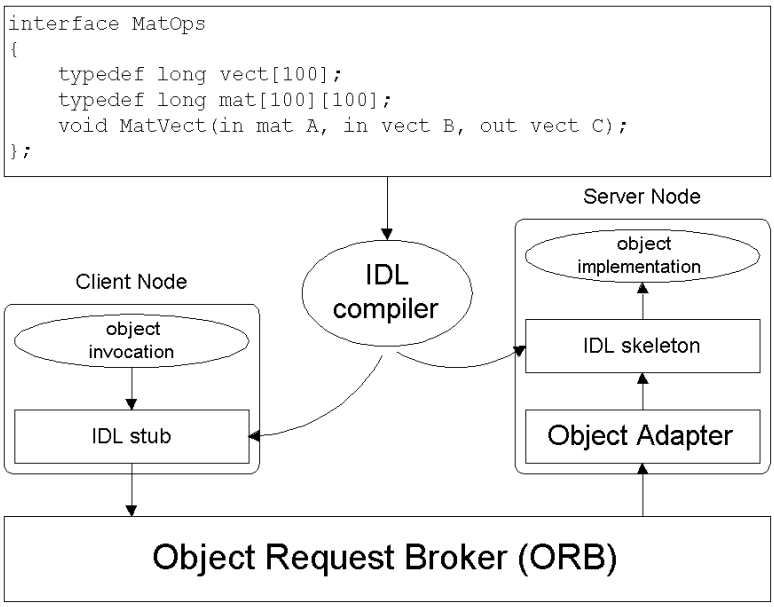

Рис. 2. Объект в технологии CORBA

```
};

class  IDomain_i : public virtual POA_Domain::IDomain {
private:
    typedef ObjectList<CORBA::Long, IElement, IElement_ptr, ElementSequence, IntSequence> ElementList;

protected:
    ElementList m_Elements;
    ...
public: virtual CORBA::Long NumberOfElements ();
virtual Domain::IElement_ptr ElementItem (CORBA::Long tag);
    ...
};

Domain::IElement_ptr IDomain_i::ElementItem (CORBA::Long tag)
  {
    return m_Elements.Item(tag);
  }
```

Рассмотрим некоторые варианты реализации распределенных данных и параллельных процессов на основе объектов CORBA.

4.2. Распределенные данные. Объекты CORBA предназначены для создания распределенных приложений, поэтому отлично подходят и для моделирования распределенных данных. CORBA-объекты одного типа запускаются многократно в разных узлах ВС и обеспечивают доступ к части распределенных данных. Для синхронизации друг с другом они содержат перекрестные ссылки, которые обеспечивают объектно-ориентированное взаимодействие между ними. Ссылки на распределенные объекты-данные можно использовать в визуальной среде управления вычислениями, в вычислительных объектах, в том числе и параллельных. Ниже приведен IDL интерфейс распределенной системы для МДО.

Данные в МДО распределяются следующим образом. Все внутренние узлы (`INode`) включаются в состав соответствующих подобластей (`ISubDomain`); внешние узлы, входящие в границы между подобластями, могут быть созданы в любой из граничащих подобластей, а остальные получают ссылки на них, как на удаленные объекты CORBA. В итоге, каждая из подобластей содержит массивы внутренних и внешних узлов, к которым может обращаться как к обычным объектам, несмотря на то, что в действительности часть из них являются заместителями (стабами). Такой подход обеспечивает более простой механизм синхронизации данных, когда каждый объект существует в одном экземпляре. Путем наследования от объекта “область” определяется разделенная область `IPartitionedDomain`, описывающая граф своих подобластей. Кроме этого, здесь представлен интерфейс делителей области. Путем различных реализаций этого интерфейса можно включить в систему ряд алгоритмов разбиения области.

CORBA — технология распределенных объектов, поэтому позволяет легко описать распределение данных в процессе вычислений. В системах, не располагающих понятиями распределенных объектов, требуется вручную программировать процессы синхронизации дублированных данных.

4.3. Параллельные процессы. Рассмотрим два способа реализации параллельных процессов: с помощью событий и асинхронных вызовов методов. Проиллюстрируем эти способы на примере объектов `IDomainSolver`, которые многократно запускаются на всех узлах вычислительной сети, соответствующих подобластям, и выполняют методы `SolveInternalX` параллельно.

Первый способ заключается в определении для параллельных процессов двух объектов: основной объект соответствует одному из процессов и включает все методы, определенные для него в объектно-ориентированной модели, причем, параллельно запускаемые методы содержат параметр-ссылку на объект-обработчик, не возвращают результат и имеют флаг `oneway` (без уведомления о выполнении); объект-обработчик содержит только `oneway`-парные методы, входными параметрами которых являются выходные параметры соответствующих методов основного объекта. Для наглядности предположим, что метод `SolveInternalX` возвращает результат типа `int`.

```
module Domain {
    interface ISubDomain : IDomain, IElement {
        readonly attribute Domain::NodeSequence InternalNodes;
        readonly attribute Numeric::IVector LastExternalResponse;
        attribute long AnalysisElementTag;
        readonly attribute Numeric::IMatrix Tangent;
        readonly attribute double Cost;

        long AddExternalNode (in Domain::INode node);
        void Changed ();
        ...
    };
    typedef sequence <Domain::ISubDomain> SubDomainSequence;

    interface IDomainPartitioner ;

    interface IPartitionedDomain : IDomain {
        readonly attribute long NumberOfSubDomains;
        readonly attribute Domain::SubDomainSequence SubDomains;
        readonly attribute Graph::IGraph SubDomainGraph;
        readonly attribute Domain::IDomainPartitioner Partitioner;
        readonly attribute Graph::IGraph DualElementGraph;
        void Partition (in long NumParts);
        long AddSubDomain (in Domain::ISubDomain subdomain);
        Domain::ISubDomain SubDomainItem (in long tag);
        Domain::INode RemoveExternalNode (in long tag);
    };
    interface IDomainPartitioner : Utilities::IDestroyedObject {
        attribute Domain::IPartitionedDomain PartionedDomain;
        readonly attribute long NumberOfPartitions;
        readonly attribute Graph::IGraph PartitionGraph;
        readonly attribute Graph::IGraph ColoredGraph;
        void Partition (in long numparts);
        void SwapVertex (in long from, in long to, in long vertextag, in boolean IsAdjacent);
        void SwapBoundary (in long from, in long to,  in boolean IsAdjacent);
        void ReleaseVertex (in long from, in long vertextag, in Graph::IGraph graph,
            in boolean IsReleaseLighter, in double greater, in boolean IsAdjacent);
        void ReleaseBoundary (in long from, in Graph::IGraph graph,
            in boolean isreleaselighter, in double greater, in boolean IsAdjacent);
        void Balance (in Graph::IGraph graph);
    };
};

module SOE {
    interface IDomainSolver : ILinearSolver {
        ...
        void SolveInternalX ();
        void SolveExternalX ();
    };
};


module SOE {
```

```
    interface IDomainSolver : ILinearSolver {
        ...
        oneway void SolveInternalX (IDomainSolverHandler Handler);
    };
    interface IDomainSolverHandler {
        void SolveInternalX (int Result);
    };
};
```

Набор активированных основных объектов после выполнения того или иного параллельного метода возвращает результат, вызывая парный метод обработчика и передавая ему в качестве входных параметров результаты своей работы. Объект-обработчик предназначен для использования в клиентской части программы, в которой одновременно вызывается параллельный метод у набора основных объектов, и не ожидает завершения их выполнения. Далее, периодически прерываясь на обработку результата, полученного от какого-либо процесса, клиентский код может выполняться до тех пор, пока не понадобятся результаты параллельного метода. В итоге все объекты-процессы функционируют параллельно, асинхронно клиентскому приложению. Ниже приведены фрагменты серверного и клиентского приложений.

Второй способ реализуется на CORBA платформах с интерфейсом AMI (Asynchronous Method Invocation — асинхронный вызов методов). AMI описывает два способа асинхронного вызова методов: событийный (`callback`) и опроса (`polling`). В событийном механизме клиентское приложение асинхронно вызывает тот или иной метод CORBA-объекта и передает ему в качестве входного параметра ссылку на обработчик события, который уже активирован на клиентской стороне. По окончании выполнения операции брокер объектов вызывает парный метод обработчика и передает ему в качестве входных параметров результаты операции. По сути, это автоматизированный аналог изложенного выше подхода. В данном случае нет необходимости вводить вспомогательные CORBA-объекты. IDL-компилятор генерирует дополнительный код автоматически, а ПО CORBA обеспечивает взаимосвязь серверного объекта и обработчика событий.

Механизм опроса (`polling`) позволяет программировать непрерывный процесс, содержащий параллельные вызовы. Обработчики асинхронных методов приостанавливают выполнение клиентского приложения в тот момент, когда он нуждается в результатах выполнения метода, до того момента, когда метод будет выполнен. В таком случае нет необходимости дробить весь вычислительный процесс на код до асинхронного вызова, код обработки событий, связанных с завершением выполнения одной из нитей, и код заключительной обработки.

```
class  IDomainSolver_i : public virtual POA_SOE:: IDomainSolver_i
{
    ...
public: virtual void SolveInternalX (SOE::IDomainSolverHandler_ptr Handler);
    ...
};
void IDomainSolver_i::SolveInternalX (SOE::IDomainSolverHandler_ptr Handler)
  {
        int Result = ...;
        Handler-> SolveInternalX(Result);
  }
```

В обоих случаях IDL-компилятор генерирует коды основного объекта и обработчика (событийного или опросного) и готовые к использованию клиентские стабы с асинхронными вызовами. Асинхронные вызовы обозначаются предикатами `sendc_/sendp_` для `callback/polling` методов соответственно. В этом случае нет необходимости вводить вспомогательные интерфейсы и объекты; кроме этого, AMI обеспечивает методы обработки ошибок.

Технология CORBA обеспечивает различные механизмы взаимодействия с объектами, что позволяет описать параллельные процессы на основе событий. Может показаться, что событийные описания несколько громоздки, но зато они связаны с объектами, что позволяет динамически создавать и конфигурировать процессы. Традиционная программа MPI-1 вообще не имеет средств динамического запуска процессов, в MPI-2 введены триды. Наличие в промежуточном ПО таких средств позволяет увеличить производительность и обеспечивает универсальный платформо-независимый инструментарий для разработки программ.

4.4. Использование прикладных C++ библиотек. Реализация разделителей графа на основе Metis, заключается в конвертировании входных данных из объектов `Graph` модели МКЭ в массивы Metis и

в обратном конвертировании выходных данных. Эта функция выполняется CORBA-объектом разделителем области `MetisPartitioner`, получаемый путем наследования от CORBA-объекта `DomainPartitioner`. `MetisPartitioner` собирает данные со всех подобластей, находящихся на разных вычислительных узлах, и во время выполнения существует в системе в одном экземпляре. Недостатком такого подхода является то, что последовательное разбиение графа в реальных задачах затратно относительно обменов между вычислительными узлами, а также требует значительных ресурсов от узла, на котором выполняется разбиение.

```
class  IDomainSolverHandler_i : public virtual POA_SOE::IDomainSolverHandler
{
    ...
public: virtual void SolveInternalX (int Result);
    ...
};
void IDomainSolverHandler_i::SolveInternalX (int Result)
  {
        ...
        Received++;
  }
int Received = 0;

int main(int argc, char** argv) {
    ...
    //соединение с серверными объектами SubDomain/DomainSolver/LinearSOE/
      IncrementalIntegrator и размещение их в массивах
      SubDomains/DomainSolvers/LinearSOEs/IncrementalIntegrators размера int Count
    ...
    for(int i=0; i<Count; i++)
    {
        IVector_i *extResponse = SubDomains[i]->LastExternalResponsePtr();
        DomainSolvers[i]->SetComputedExternalX(extResponse);

        IDomainSolverHandler_i *Handler = new IDomainSolverHandler_i();
        DomainSolvers[i]->SolveInternalX(Handler->this());
    }
    while(Received < Count)
    {
    ...
    }
    for(int i=0; i<Count; i++)
    {
        IVector_i *X = LinearSOEs[i]->XPtr(); IncrementalIntegrators[i]->UpdateState(X);
    }
    ...
}
```

Технология CORBA ориентирована на создание распределенных объектно-ориентированных приложений. Для повышения производительности в стандарт введены различные схемы взаимодействия с объектами, интерфейс AMI. Основной проблемой стандарта является практически полное отсутствие инструментов установки и управления объектами в вычислительной сети. Решением является развитие компонентной модели, которая обеспечивает не только управление CORBA-объектами, но и более высокоуровневые средства разработки приложений.

5. Технология параллельных распределенных компонентов. В настоящее время в стадии разработки находится ряд технологий, построенных на основе MPI и CORBA (подробнее см. в [6]). Тесты производительности [7] и анализ возможностей проектирования программ приводят к выводу, что наиболее перспективными являются проекты, связанные с развитием стандарта CORBA и его интеграцией с MPI. Среди них можно отметить технологию объектных групп OGS-JS (`Object Group Service` — сервис объектных групп, `Join Service` — сервис объединений) [8], основанную на событийном механизме и методах без уведомления, а также проект по разработке параллельной распределенной модели [9] на основе CCM и MPI.

В данной работе предлагается подход для развития технологии CORBA в области компонентного проектирования программ и высокопроизводительных вычислений — технология параллельных распределенных компонентов. Основными ее составляющими являются:

— компонентная система;

— распределенная компонентная система;

— параллельная компонентная система.

В основу технологии положены стандарт компонентной модели CORBA CCM (CORBA Component Model) [10], а также интерфейс асинхронного вызова методов AMI и интеграция с MPI. Технология параллельных распределенных компонентов реализовывается на базе промежуточного ПО TAO (The ACE ORB) [11].

```
//IDL
module SOE {
    interface IDomainSolver : ILinearSolver {
        ...
        int SolveInternalX ();
        void SolveExternalX ();
    };
};
//серверная часть
class  IDomainSolver_i : public virtual POA_SOE:: IDomainSolver_i
{
    ...
public: virtual int SolveInternalX ();
    ...
};

int IDomainSolver_i::SolveInternalX ()
  {
        int Result = ...;
        return Result;
  }
//клиентская часть
class  IDomainSolverHandler_i : public virtual POA_SOE::AMI_IDomainSolverHandler
{
    ...
public: virtual void SolveInternalX (int ami_return_val);

virtual void SolveInternalX_excep (SOE::AMI_IDomainSolverExceptionHolder * excep_holder);
    ...
};

void IDomainSolverHandler_i::SolveInternalX (int ami_return_val)
  {
        ...
        Received++;
  }

int Received = 0;

int main(int argc, char** argv) {
    ...
    //соединение с серверными объектами SubDomain/DomainSolver/LinearSOE/
    IncrementalIntegrator и размещение их в массивах SubDomains/
    DomainSolvers/LinearSOEs/IncrementalIntegrators размера int Count
    ...
    for(int i=0; i<Count; i++)
    {
        IVector_i *extResponse = SubDomains[i]->LastExternalResponsePtr();
        DomainSolvers[i]->SetComputedExternalX(extResponse);

        IDomainSolverHandler_i *Handler = new IDomainSolverHandler_i();
        DomainSolvers[i]->sendc_SolveInternalX(Handler->this());
    }
    while(Received < Count)
    {
    ...
    }
    for(int i=0; i<Count; i++)
    {
        IVector_i *X = LinearSOEs[i]->XPtr(); IncrementalIntegrators[i]->UpdateState(X);
    }
    ...
}
```

5.1. Компонентная система. Компонентная система представляет собой объектно-ориентирован-

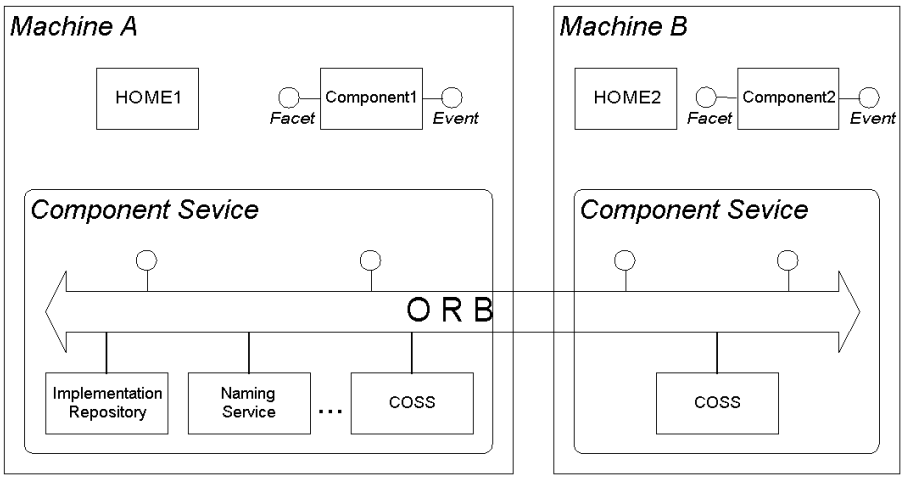

Рис. 3. Компонентная модель

ную модель программной системы. Компонентная система обеспечивает основные методы работы с программными компонентами: проектирование, установка, управление в режиме реального времени (запуск, отслеживание состояния, остановка). В качестве программных компонентов в данном случае понимаются CORBA-приложения. Сама компонентная система также реализовывается в виде CORBA-объектов, т.к. в дальнейшем она используется в составе распределенной компонентной системы.

В настоящий момент для CORBA разрабатывается стандарт компонентной модели CCM, который описывает весь жизненный цикл программных компонентов: разработка, установка, выполнение. В связи с отсутствием полноценных реализаций CCM на основе `CCM::Components` создана собственная компонентная система.

Компонентная система основывается на ряде стандартных возможностей CORBA: на разработке объектов с помощью IDL и на управлении объектами с помощью сервисов (`Implementation Repository`, `Life Cycle Service`, `Naming Service`, `Event/Notification Service`). Компонентная система состоит из вспомогательной C++ библиотеки, где реализованы компоненты (объекты CCM), а также сервиса приложений, позволяющего манипулировать этими компонентами. Объекты CCM являются обычными объектами CORBA, они реализуют ряд функций, которых достаточно для того, чтобы с помощью сервиса приложений запускать и искать их экземпляры. Процесс реализации компонентов состоит из следующих этапов:

— создание интерфейсов прикладного объекта на языке IDL;

— создание прикладных объектов CORBA, реализующих интерфейсы, полученные на первом этапе;

— создание компонента путем наследования от классов библиотеки CCM-Components и прикладного объекта, реализованного на втором этапе.

Прикладной CORBA-объект с интерфейсом `Facet` погружается в шаблон `Component`, который обеспечивает создание экземпляров компонента. Конструктором компонентов выступает фабрика компонентов `Home`. На стороне клиента в шаблон, основанный на стабе `Component`, погружается CORBA-объект с интерфейсом `Event` (рис. 3).

Так как компоненты являются обычными CORBA-объектами, то целесообразно интегрировать сервис компонентов со стандартными сервисами CORBA, чтобы использовать инструменты для управления объектами. `Implementation Repository` используется для активации компонентов, `Naming Service` обеспечивает соединение с экземплярами объектов.

5.2. Распределенная компонентная система. Распределенная компонентная система описывает и реализует работу с компонентами, распределенными по вычислительной сети. Реализация распределенной системы агрегирует реализацию компонентной системы, позволяя взаимодействовать нескольким компонентным системам, сопоставленным с узлами вычислительной системы.

Основным направлением в развитии распределенной модели является создание описания архитектуры программно-аппаратной среды и специальных служб, которые в режиме реального времени выдают состояние вычислительных узлов и коммуникаций (степень загруженности процессоров, объемы занятой и свободной оперативной памяти, объемы пересылок и т.д.).

5.3. Параллельная компонентная система. Наиболее важной частью вычислительной системы является технология параллельных распределенных вычислений, которая является расширением технологии асинхронного вызова методов AMI и создается на базе ПО TAO. В TAO частично реализован интерфейс AMI, в частности событийный механизм, поэтому технология параллельных компонентов в первую очередь основана на этом способе асинхронного вызова методов. В будущем, планируется использовать и механизм опроса.

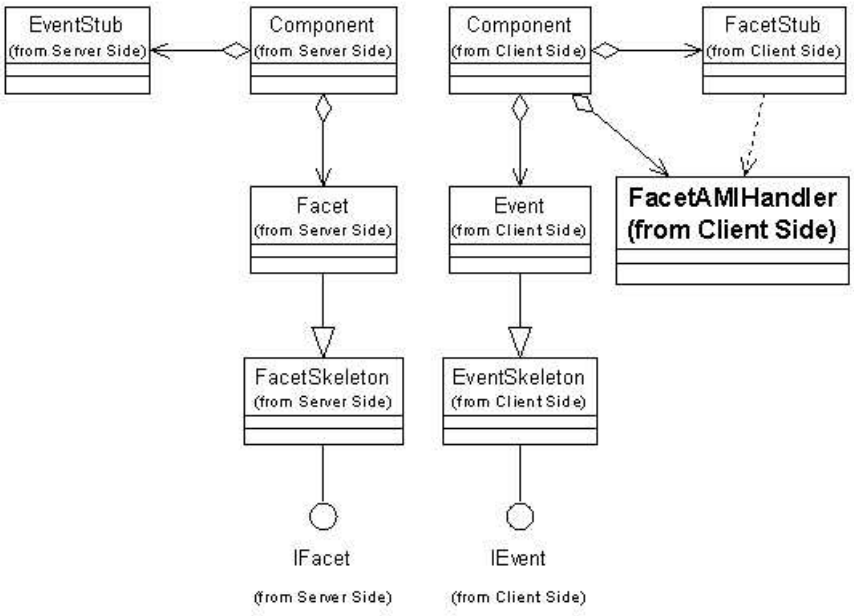

Рис. 4. Структура параллельного компонента

Структура параллельного компонента отличается тем, что на клиентской стороне вводится обработчик событий AMI (рис. 4). При такой структуре возможны три способа взаимодействия с компонентом:

— синхронный вызов методов интерфейса `Facet` посредством `FacetStub`;

— обработка событий, определенных интерфейсом `Event`, посредством CORBA-объекта `Event`;

— асинхронный вызов методов интерфейса `Facet` посредством `FacetStub` и последующая обработка событий посредством `FacetAMIHandler`.

5.4. Интеграция MPI и CORBA. Необходимость интеграции промежуточной системы MPI обусловлена ее популярностью в мире параллельных вычислений и большому количеству программного обеспечения, разработанного на ее основе. В данной работе используется промежуточное ПО MPICH [12].

Инкапсуляция MPI-кода может быть осуществлена посредством клиентских или серверных CORBA-приложений (рис. 5). Первый способ предназначен для инкапсуляции ППП, основанных на технологии MPI. Он заключается в реализации клиентского приложения, содержащего MPI-код, и CORBA-объекта, синхронизирующего набор одновременно запускаемых клиентских приложений. В данном случае общий и распределенный объектно-ориентированный интерфейс к ППП реализованы в CORBA-объекте, параллельное поведение не имеет каких-либо объектно-ориентированных описаний и реализовано в клиентских приложениях. Использование этого подхода будет продемонстрировано ниже на примере библиотеки ParMetis.

Второй способ предназначен для создания прикладных CORBA-объектов, имеющих параллельное поведение на основе MPI. Можно выделить два варианта реализации этого способа: многократный запуск CORBA-серверов в контексте MPI и выполнение в контексте MPI только методов многократно запущенных CORBA-серверов в контексте MPI. Кроме этого, возможны варианты с использованием MPI-тридов. MPI обеспечивает параллельные обмены между одновременно запущенными объектами, а CORBA — объектно-ориентированный интерфейс параллельных распределенных вычислений. Этот подход обеспечивает более широкие возможности для инкапсуляции ППП на основе MPI, позволяя создать

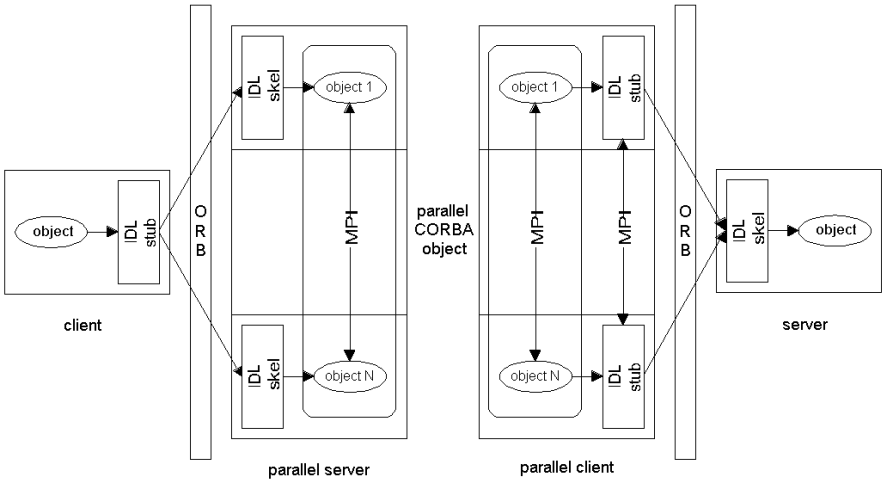

Рис. 5. Интеграция MPI и CORBA

не только общий объектно-ориентированный интерфейс ППП, но еще и параллельный распределенный. CORBA-приложение может взаимодействовать с отдельными параллельными компонентами-процессами, которые инкапсулируют ППП.

В технологию параллельных распределенных компонентов включена интеграция с MPI-методом многократного запуска серверных приложений в контексте MPI. Для этого реализованы CORBA-сервис `MPIComponentService`, позволяющий конфигурировать систему MPI и многократно запускать серверные приложения в контексте MPI, и C++ библиотека, содержащая базовый класс `MPIComponent`.

Технология параллельных распределенных компонентов обобщает методы проектирования объектно-ориентированных программ, применяемые в технологиях MPI и CORBA. Новым является компонентный подход, обеспечивающий объектно-ориентированную модель вычислительной системы. Компонентная модель реализована в виде промежуточного ПО (сервис и библиотека). Компонентная модель и промежуточное ПО расширены для проектирования параллельных распределенных программ.

5.5. Реализация МДО на основе технологии параллельных распределенных компонентов. При реализации метода декомпозиции в системе для конечно-элементного анализа на основе технологии параллельных распределенных компонентов используются основные свойства этой технологии: компонентный подход и параллельные вычисления. В данной реализации используются уже созданные прикладные CORBA-объекты, при этом модифицировать придется лишь те из них, которые должны совершать асинхронные вызовы.

Для создания компонента необходим прикладной CORBA-объект и, если надо, CORBA-объект обработки событий. Компонент получается путем агрегации в шаблон `Component` прикладного объекта и, если предусмотрено, стаба объекта обработки событий. Взаимодействие с компонентом обеспечивает стаб `Component`, в который агрегированы стаб прикладного объекта, обработчик AMI и, если необходимо, объект обработки событий.

Моделирование распределенных данных с помощью параллельных распределенных компонентов полностью базируется на технологии CORBA. Моделирование параллельных процессов основано на использовании шаблонов параллельных компонентов, обеспеченных технологией параллельных распределенных компонентов.

Рассмотрим интеграцию MPI-кода в виде клиентских CORBA-приложений на примере библиотеки ParMetis. Разделители графов состоят из управляющего приложения ParMetis-сервис, который включает в себя CORBA-сервер, и подчиненного приложения ParMetis. В процессе вызова метода разделения графа ParMetis-сервис запускает несколько экземпляров клиентских приложений под управлением MPI. Каждый экземпляр ParMetis соответствует отдельной подобласти, забирает у сервера часть графа, конвертирует его в массивы в формате ParMetis, производит операцию разбиения в рамках MPI совместно с другими приложениями, конвертирует обратно результат разбиения, передает результат сервису и завершает работу.

```
int main(int argc, char *argv[]) {
// инициализация MPI
```

```
    int myid, numprocs;
    int namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Comm comm;

    MPI_Init(&argc,&argv);

    MPI_Comm_dup(MPI_COMM_WORLD, &comm);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    try
    {
// инициализация CORBA
        ...
// соединение с сервисом IParMETISSystem
        CosNaming::Name name (1);
        name.length (1);
        name[0].id = CORBA::string_dup ("ParMETIS");
        CORBA::Object_var ParMETISSystem_object = naming_context->resolve (name);

        MPI_Barrier(comm);

        ParMETIS::IParMETISSystem_var ParmetisSystem;
        ParmetisSystem = ParMETIS::IParMETISSystem::_narrow(ParMETISSystem_object.in());
// получение порции данных IParMETISData от IParMETISSystem
        ParMETIS::IParMETISData_var ParMetisData;
        ParmetisSystem->GetParMETISData(CORBA::UShort(myid), ParMetisData.out());
// метода разбиения графа
        ParMETIS::ParMETISMethod Method;
        ParmetisSystem->GetMethod(Method);
// часть графа
        ParMETIS::ParMETISGraph_var Graph;
        ParMetisData->GetGraph(Method, Graph.out());
// разбиение графа (в зависимости от заданного метода)
        switch (Method)
        {
//******PartKway***************
        case ParMETIS::PartKway:
            {
// перевод данных в формат ParMETIS
                ...
// MPI вызов функции ParMETIS_PartKway
                ParMETIS_PartKway(vtxdist, xadj, adjncy, vwgt, adjwgt,
                &wgtflag, &numflag, &nparts, options, &edgecut, part, &comm);
// формирование нового графа
                Graph->ECut=CORBA::Long(edgecut);
                Graph->Prt.length(Size+1);
                for (i=0; i<Size; i++)
                {
                    Graph->Prt[i]=CORBA::Long(part[i]);
                }
// return results
                ParMetisData->SetGraph(Method, Graph.in());
                ...
            }
            break;
        ...
        }
// деактивация CORBA
        ...
    }
    catch (CORBA::Exception &)
    {
        cerr << "CORBA exception raised!" << endl;
    }
// деактивация MPI
    MPI_Finalize();     return 0;
}
```

В настоящее время ведется работа по созданию параллельной распределенной объектно-ориентиро-

ванной модели ParMetis, которая будет реализована на основе технологии интеграции MPI-кода в виде серверных приложений в рамках технологии параллельных распределенных компонентов.

7. Заключение. В данной работе предложены расширения стандарта CORBA, ориентированные на высокопроизводительные вычисления. На сегодня CORBA предоставляет шину для взаимодействия объектов и службы для их сопровождения. На этой базе можно создать технологию построения сложных высокопроизводительных вычислительных систем, реализующих те или иные математические модели. Одним из решений является технология параллельных распределенных компонентов.

Компонентный подход, реализуемый в вычислительной системе, обеспечивает простой механизм разработки сложных приложений, позволяя описать объекты прикладной задачи и компоненты программной системы. Кроме этого, данный подход позволяет интегрировать в прикладную систему открытое программное обеспечение: CAD-системы, математические библиотеки и т. д.

Проблема больших вычислительных затрат в прикладных задачах решается с помощью технологий параллельных и распределенных вычислений. Основными составляющими высокопроизводительных вычислений в разрабатываемой системе являются описания аппаратно-программной платформы, распределенных данных и параллельных процессов. Эти описания реализуются с помощью стандартных средств CORBA, а также путем интеграции существующих технологий, таких как MPI.

В дальнейшем предполагается развивать систему параллельных распределенных компонентов по ряду направлений, среди которых можно выделить систему балансировки нагрузки, реализующую модель описания производительности и производящую более или менее автоматическое выравнивание нагрузки.

Работа выполнена при поддержке Российского фонда фундаментальных исследований (проект 02–07–90265) и Уральского отделения РАН (грант для молодых ученых).

СПИСОК ЛИТЕРАТУРЫ

1. *Копысов С.П., Красноперов И.В., Рычков В.Н.* Объектно-ориентированный метод декомпозиции области // Вычислительные методы и программирование. 2003. **1**, № 1. 1–18.
2. *Рычков В.Н., Красноперов И.В., Копысов С.П.* Объектно-ориентированная параллельная распределенная система для конечно-элементного анализа // Матем. моделирование. 2002. **14**, № 9. 81–86.
3. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. М.: Бином, 1999.
4. *Лисков Б., Гатэг Дж.* Использование абстракций и спецификаций при разработке программ. М.: Мир, 1989.
5. *Копысов С.П., Новиков А.К.* Параллельные алгоритмы перестроения и разделения неструктурированных сеток // Матем. моделирование. 2002. **14**, № 9. 91–96.
6. *Рычков В.Н., Красноперов И.В., Копысов С.П.* Промежуточное ПО для высокопроизводительных вычислений //Вычислительные методы и программирование. 2001. **2**, № 2. 117–132.
7. *Denis A., Pérez C., Priol T.* Towards high performance CORBA and MPI middlewares for grid computing. INRIA Rapport de Rechercher, N 4555. 2002.
8. *Aleksy M., Korthaus A.* A CORBA-based object group service and join service providing a transparent solution for parallel programming // Proc. of the Intern. Symposium on Software Engineering for Parallel and Distributed Systems, IEEE. 2000.
9. *Pérez C., Priol T., Ribes A.A.* Parallel CORBA component model. INRIA Rapport de Rechercher, N 4552. 2002.
10. OMG. CORBA 3.0. New Component Chapters. CCM FTF Draft ptc/99-10-04 (`http://www.omg.org`).
11. Real-time CORBA with TAO$^{(TM)}$ (The ACE ORB) (`http://www.cs.wustl.edu/~schmidt/TAO.html`).
12. *Ashton D., Chan A., Gropp B., Lusk R., Swider D., Thakur R.* Portable MPI Model Implementation. Version 1.2.0. Argonne National Laboratory. 1999 (`http://www.mcs.anl.gov/mpi/mpich`).

Поступила в редакцию
28.02.2003