УДК 681.3.06

ПРОМЕЖУТОЧНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ДЛЯ ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ВЫЧИСЛЕНИЙ

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

Рассматриваются технологии высокопроизводительных вычислений. Определяется структура вычислительной системы. Проводится анализ существующих систем промежуточного программного обеспечения для параллельных и распределенных вычислений. Формулируются требования к интегрированной платформонезависимой системе, реализующей технологию параллельных распределенных компонентов.

1. Введение. Повышение сложности математических моделей требует высокой производительности аппаратных и программных вычислительных средств. Многопроцессорные архитектуры являются одним из способов повышения производительности аппаратного обеспечения, а модели параллельных и распределенных вычислений — программного обеспечения. Основной проблемой сегодня становится соответствие аппаратного и программного обеспечения. Введение промежуточного программного обеспечения, которое связывает ЭВМ и прикладные программы, является одним из возможных решений.

1.1. Структура вычислительной системы. Под вычислительной системой (ВС) будем понимать множество элементов аппаратного (hardware), промежуточного программного (middleware) и прикладного программного (software) обеспечения с набором связей между ними. Все эти элементы объединены с целью проведения вычислений под управлением наблюдателя. Данное понятие полностью соответствует общей теории вычислительных систем [1].

Взаимодействие элементов в ВС строится следующим образом. С элементами аппаратного обеспечения непосредственно взаимодействуют элементы промежуточного, которые, учитывая все особенности архитектуры ЭВМ, предоставляют в совокупности программные средства управления ВС (программную модель ВС). Прикладные программы software через программную модель, обеспеченную middleware, реализуют те или иные алгоритмы для решения прикладной задачи, которая является целью данной ВС. Необходимо отметить, что промежуточное программное обеспечение (ПО) может быть многоуровневым, когда элементы объединяются в некоторые подсистемы, которые находятся в иерархической зависимости, например, операционная система – коммуникационная среда – система распределенных компонентов.

1.2. Программная модель ВС. Программная модель высокопроизводительной ВС включает общий интерфейс доступа к аппаратным средствам и описание параллельных процессов над распределенными данными. В связи с использованием мультипроцессорных и сетевых архитектур общий интерфейс аппаратных средств включает в себя описание вычислительного узла (процессора в многопроцессорной ЭВМ или рабочей станции в компьютерной сети) и коммуникации между узлами (шина обмена данными или сетевое соединение). Далее в одних системах (MPI) интерфейс расширяется для построения сети процессоров (учитывается топология соединений), в других (CORBA) — для оптимизации взаимодействий (учитываются характеристики среды обмена).

Для организации высокопроизводительных вычислений применяются различные способы распараллеливания. Наиболее удобной является классификация моделей ВС, которая базируется на понятии потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. На основе числа потоков команд и потоков данных рассмотрим три модели: MISD (Multiple Instructions Single Data), SIMD (Single Instructions Multiple Data), MIMD (Multiple Instructions Multiple Data) [2]. ВС, основанные на SIMD-модели, сегодня принято называть параллельными; ВС, основанные на MISD-модели, — распределенными. Стандартом параллельных вычислений стала де-факто технология MPI (Message Passing Interface — интерфейс передачи сообщений), стандартом распределенных вычислений общепризнанна технология CORBA (Common Object Request Broker Architecture — общая архитектура брокера объектных запросов).

Ниже будут проанализированы различные системы промежуточного ПО и сформулирован ряд требований, выдвигаемых к программной модели высокопроизводительной ВС.

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

2. Промежуточное ПО для параллельных вычислений. Главной целью параллельных вычислений является существенное уменьшение времени расчета задач, связанных с большими объемами данных. Параллельные вычисления подразумевают единый вычислительный процесс, запущенный над параллельными данными. В основе этого подхода лежит методика обработки данных и мало внимания уделяется организации процессов, поэтому главной проблемой параллельных прикладных пакетов становится сложность текстов. Параллельное программирование превращается в объединение различных процессов в одну программу. Код параллельных программ плохо структурируется, появляется необходимость введения специальных конструкций для дифференциации действий в рамках одного процесса.

Примерами параллельных технологий являются PVM (Parallel Virtual Machine — параллельная виртуальная машина) [3] и MPI [4, 5]. PVM является одним из первых SIMD-интерфейсов. Технология MPI появилась на его основе, оказалась более популярной и была реализована для разных типов ЭВМ. Во многом они схожи, поэтому далее будет рассматриваться только система MPI.

2.1. SIMD-интерфейс с обменом сообщениями MPI. Парадигма MPI является развитием идеи последовательного программирования для параллельных вычислений. Несколько задач последовательного программирования рассматриваются вместе, для их кооперации используются средства коммуникации. Таким образом, MPI имеет интерфейс передачи сообщений, но не имеет интерфейсов общего адресного пространства или непосредственного обращения одного процессора к памяти другого.

Эффективность MPI достигается также за счет отсутствия ненужной работы по пересылке лишней информации с каждым сообщением или кодированию-декодированию заголовков сообщений. MPI был разработан так, чтобы поддерживать одновременное выполнение вычислений и коммуникаций. Это достигается использованием неблокируемых коммуникационных вызовов, которые разделяют инициацию коммуникации и ее завершение. Для повышения скорости в MPI используются и другие приемы. Например, встроенная буферизация позволяет избежать задержек при отправке данных — управление в передающую ветвь возвращается немедленно, даже если ветвь-получатель еще не подготовилась к приему. MPI использует многопоточность (multithreading), вынося большую часть своей работы в потоки (threads) с низким приоритетом. Буферизация и многопоточность сводят к минимуму негативное влияние неизбежных простоев при пересылках на производительность прикладной программы. На передачу данных типа один–всем затрачивается время, пропорциональное не числу участвующих ветвей, а логарифму этого числа. Перечислим основные особенности MPI.

1) *Взаимодействие точка-точка.* Основной механизм коммуникаций в MPI — передача данных между парой процессов: одна сторона посылает, другая получает. MPI обеспечивает набор функций посылки и приема, которые передают данные определенного типа с ассоциированным идентификатором сообщения. Существуют два вида функций для коммуникаций типа точка-точка. Это блокирующие и неблокирующие вызовы.

2) *Коллективные операции.* Под термином "коллективные" в MPI подразумеваются три группы функций: функции коллективного обмена данными, точки синхронизации (или барьеры) и функции поддержки распределенных операций. Коллективная функция одним из аргументов получает описатель области связи (коммуникатор). Вызов коллективной функции является корректным, только если тот произведен из всех процессов соответствующей области связи и именно с упомянутым коммуникатором в качестве аргумента (хотя для одной области связи может иметься несколько коммуникаторов, подставлять их вместо друг друга нельзя). В этом и заключается коллективность: либо функция вызывается всем коллективом процессов, либо ни одним из них.

3) *Группы процессов.* Группа — это некое множество ветвей. Одна ветвь может быть членом нескольких групп. В распоряжение программиста предоставлен тип MPI_GROUP и набор функций, работающих с переменными и константами этого типа. Констант, собственно, две: MPI_GROUP_EMPTY может быть возвращена, если группа с запрашиваемыми характеристиками в принципе может быть создана, но пока не содержит ни одной ветви; MPI_GROUP_NULL возвращается, когда запрашиваемые характеристики противоречивы. Согласно концепции MPI, после создания группу нельзя дополнить или уменьшить, однако можно создать только новую группу под требуемый набор ветвей на базе существующей. С помощью групп процессов обеспечиваются групповые коммуникации.

4) *Области коммуникаций* (communication domain) — это абстрактное понятие: в распоряжении программиста нет типа данных, описывающего непосредственно области связи, как нет и функций по управлению ими. Области связи автоматически создаются и уничтожаются вместе с коммуникаторами. В одну область связи входят все задачи одной или двух групп. Для каждой области связи существует коммуникатор (ее описатель). Именно с коммуникаторами программист имеет дело, вызывая функции пересылки данных, а также подавляющую часть вспомогательных функций. Одной области связи могут

соответствовать несколько коммуникаторов. Коммуникаторы являются "несообщающимися сосудами": если данные отправлены через один коммуникатор, ветвь-получатель сможет принять их только через тот же самый коммуникатор, но ни через какой-либо другой.

5) *Топологии процессов*. MPI позволяет пользователю определять топологию процессов, т.е. то, как они будут размещаться на вычислительных узлах и каким образом будут происходить коммуникации между ними. Правильный выбор топологии может значительно уменьшить время, затрачиваемое на пересылку данных между процессами.

К сожалению, интерфейс MPI совершенно не поддерживает объектно-ориентированный подход (ООП), поэтому прикладные программы, основанные на данном middleware, слишком сложны, а ВС не обладают свойствами гибкости и расширяемости. Излишняя простота самого интерфейса, с одной стороны, удобна, т.к. нет необходимости в длительном его освоении, но, с другой стороны, предполагает необходимость использования других библиотек, в том числе платформозависимых, как, например, различные библиотеки, реализующие многопоточность, что лишает ВС необходимого свойства переносимости.

В настоящее время парадигма передачи сообщений является очень распространенной. Одна из причин этого — большое число платформ, поддерживающих модель передачи сообщений. Программы, написанные в стиле MPI, могут выполняться на многопроцессорных системах с общей или разделяемой памятью, на сетях рабочих станций и даже на однопроцессорных машинах. Исходный код MPI-программ переносим на любые платформы, поддерживающие модель MPI, но программы под разными платформами чаще всего взаимодействовать не могут. MPI в настоящее время стандартизован и широко используется. Существует много придерживающихся этого стандарта программных пакетов для различных ОС, таких как MPICH, WMPI и т.п.

2.2. Межплатформенная реализация MPICH-G. Особое место среди систем, реализующих интерфейс MPI, занимает MPICH-G [6]. В данной реализации решается проблема взаимодействия MPI-программ, работающих на разных платформах, а также предлагается дополнительный интерфейс для организации параллельных вычислений через Internet. Новые возможности обеспечиваются промежуточным ПО Globus/Nexus [7, 8], которое является ядром MPICH-G. Программная среда нижнего уровня обеспечивает коммуникационный интерфейс и интерфейс запуска процессов, эмулируя виртуальную машину, на которой работает MPICH-G. Это дает возможность организовать одновременный запуск и взаимодействие MPI-процессов на всех платформах, которые поддерживают Globus.

MPICH-G — пример многоуровневой системы промежуточного ПО, благодаря чему имеет интерфейс, скрывающий гетерогенность аппаратной среды и позволяющий запускать на этой архитектуре программы. MPICH-G обеспечивает хорошую переносимость программ на уровне исходного кода.

2.3. Объектно-ориентированные SIMD-интерфейсы с обменом сообщениями (MPI++, OOMPI). Первым этапом введения объектной ориентированности в параллельные вычисления на MPI является построение модели классов самой системы MPI. Существует много попыток ввести поддержку C++ в MPI. Рассмотрим реализацию двух из них — MPI++ [9] и OOMPI [10]. Эти библиотеки инкапсулируют функциональность MPI в классовую иерархию для обеспечения простого, переносимого и интуитивного интерфейса. Путем наследования пользователи могут создавать свои прикладные классы. В данных системах предложена модель, состоящая из объектов: коммуникатор, группа коммуникаторов, сообщение и некоторых других. Такой подход позволяет создавать достаточно простые, гибкие и расширяемые прикладные программы, но остается проблема разрушения объектных моделей при использовании сообщений. Чтобы избежать этого, разработчикам необходимо создавать значительное количество кода, связанного с трансляцией пересылаемых объектов в сообщения и маршалингом вызываемых методов объектов. Более подробно об этих библиотеках см. в [11].

2.4. SIMD-интерфейсы с объектными обменами. Проблема объектных обменов решена в системах TPO++ [12] и Charm++ [13]. Данные системы включают наборы классов, которые могут сопоставляться с какими-либо объектами, участвующими в межпроцессорных взаимодействиях. Путем наследования от этих классов, реализующих между собой объектные обмены на основе раннего связывания, можно создавать объекты для конкретной прикладной задачи. Объектная модель поддерживается полностью, но она достаточно бедна, поэтому предполагается, что пользователи будут расширять ее посредством наследования. К сожалению, наследование не удовлетворяет полностью требованиям гибкости программного обеспечения: при расширении системы требуется ее полная перекомпиляция.

Главным нововведением является процедура маршалинга [14], состоящая во взаимодействии с удаленным объектом. Эта процедура хорошо формализуется и может быть автоматизирована, но в данных системах ее нужно заново определять для новых объектов. Кроме того, не реализована общая модель клиент-сервер. Клиент-серверный подход подразумевает параллельную обработку серверным объектом

действий, вызываемых несколькими клиентами одновременно. Многопоточность — основа для реализации модели клиент-сервер и распределенных объектов, когда необходим динамический запуск дополнительных потоков команд на одном процессоре.

2.4.1. TPO++. Проект TPO++ — это объектно-ориентированная библиотека передачи сообщений, написанная на C++ поверх MPI. Ее главные особенности — простая передача объектов, сохранение типов (type-safety), MPI-совместимость и интегрирование с библиотекой STL (Standard Template Library — стандартная библиотека шаблонов).

Разработчики ставили перед собой следующие цели.

1) *Объектная ориентация и сохранение типов.* Главная цель объектно-ориентированной библиотеки классов, поддерживающей передачу сообщений (message-passing) — это тесная интеграция с объектно-ориентированными концепциями, в особенности передача объектов в простой, эффективной и безопасной (type-safety) манере. Решение не должно нарушать существующие возможности ООП, такие как инкапсуляция и наследование.

2) *Интеграция* с C++. Реализация на C++ должна принимать во внимание все новые возможности языка, такие как использование исключений для обработки событий и даже, что, по мнению авторов, более важно, интеграцию со стандартной библиотекой шаблонов STL (с поддержкой STL-контейнеров и соглашений об интерфейсах STL).

3) *MPI-совместимость.* Разработка должна соответствовать MPI-интерфейсу и соглашению имен, а также быть семантически настолько близкой, насколько это возможно без нарушения объектно-ориентированных концепций. Это помогает перестраивать C-программы и легко переносить существующий C- и C++-коды.

4) *Эффективность.* Реализация должна не слишком отличаться от MPI в терминах передачи сообщений и эффективности памяти. Поэтому она должна быть легкой, насколько это возможно, позволяя C++-компилятору статически убирать почти все интерфейсы верхнего уровня. Коммуникации должны быть реализованы через MPI-вызовы и, если это возможно, не использовать дополнительные буферы, которые сохраняют память, а также позволять низкоуровневой MPI-реализации оптимизировать коммуникации, например, если сетевое оборудование способно получать и посылать рассеянные блоки памяти автоматически.

Эти цели достигаются с помощью техники трейтов (trait) [15]. Техника трейтов позволяет писать общий код, определяя класс не по типу, как в простых шаблонах, а по частным характеристикам типа. Основываясь на таких характеристиках компилятор может генерировать специфический код. Главное решение заключается в том, чтобы не задавать эти характеристики как новый параметр в обычных шаблонах, который может раздражать пользователя, обычно не интересующегося деталями реализации, а вместо этого определять новый шаблон. Этот шаблон (трейт-шаблон) может использоваться для внутреннего накопления специфической информации об актуальных параметрах шаблона. Любые коммуникационные методы верхнего уровня, дающие информацию о типе, можно однозначно заменить последовательностью базовых типов. Важно отметить, что большинство преобразований статично, то есть может быть выполнено во время компиляции, поэтому можно быть уверенным в сохранении эффективности нижележащего MPI-кода.

Этой информации достаточно для запаковки данных в коммуникационный буфер или для создания MPI-типа для передачи. Реализация TPO++ адресуется только к однородным архитектурам. Никакая информация о типах не отображается в коммуникациях. Блоки памяти, полученные преобразованиями, передаются напрямую. Информация о типе, полученная во время приведения, может использоваться для быстрого построения MPI-типов данных. Если бы оптимизация для тривиальных конструкторов была опущена, то это также обобщило бы реализацию на гетерогенную архитектуру.

Хотя в этой библиотеке реализована достаточно эффективная передача данных объекта, в ней все-таки существуют много нерешенных проблем. Первая — это сложность пересылки динамически определенных пользовательских типов данных. Для этого требуется в каждом таком типе определять маршалинг его данных. Вторая, наиболее важная для распределенных систем проблема, — это удаленный вызов методов объектов. В TPO++ он не реализован. Все методы объекта вызываются только на процессоре, на котором находится сам объект. Для того чтобы вызвать его метод на другом процессоре, нужно переслать туда все его данные. В библиотеке TPO++ нет понятия распределенного объекта, что необходимо для реализации полностью объектно-ориентированной распределенной параллельной системы.

2.4.2. Charm++. В системе Charm++ для процедуры маршалинга предлагается модель актера (Actor) [16], включающая классы актера и заместителя, которые могут сопоставляться с какими-либо объектами, участвующими в межпроцессорных взаимодействиях. Путем наследования от этих классов,

реализующих между собой маршалинг на основе раннего связывания, можно создавать объекты для конкретной прикладной задачи. Charm++ построена на базе языка программирования C++. Подобно C++, Charm++-объекты могут содержать приватные данные и публичные методы [17]. Различие заключается в том, что эти методы могут вызываться с удаленных процессоров асинхронно. Асинхронный вызов метода означает, что вызвавший его не ждет, когда метод действительно выполнится и вернет результат. Поэтому Charm++ методы, названные entry методами, не имеют возвращаемых значений. Так как реальный Charm++-объект, у которого вызывается на выполнение метод, может находиться на удаленном процессоре, возможно, в другом адресном пространстве, способ адресации C++ к объекту, такой как указатель C++, не действителен в Charm++. Вместо этого существуют два способа адресации к удаленному объекту. В первом из них каждый объект имеет уникальный дескриптор, названный ChareID. Это — структура, уникально определяющая адрес объекта на параллельной машине. Отметим, что она одинакова для любых типов объекта (его класса). Второй способ — это ссылка на объект через его прокси.

Для каждого объектного типа существует его прокси (фактически генерируемый интерфейсным компилятором), который содержит ссылку, например ChareID, на реальный объект. Кроме того, методы прокси класса сообщаются с методами реального класса и действуют как опережающие. Это означает, что когда метод вызывается на прокси удаленного объекта, он переправляет этот вызов реальному объекту. Прокси возвращает результат созданием нового удаленного объекта. Все объекты, создаваемые и управляемые удаленно в Charm++, имеют прокси. Прокси для каждого объекта в Charm++ имеют некоторые различия из-за особенностей, ими поддерживаемых, но основной синтаксис и семантика почти всегда одинаковы; это позволяет вызывать методы на удаленных объектах с помощью их прокси.

Charm++-программа состоит из множества объектов, распределенных на доступных процессорах. Таким образом, основной модуль параллельных вычислений в программе — это chare-объект, который создается на любом доступном процессоре и который может обращаться к любому процессору. Chare в данной программе играет роль Actor. Эти объекты создаются динамически и их множество может действовать одновременно. Chare-объекты посылают сообщения друг другу для запуска своих методов асинхронно. Концептуально, система обрабатывает пул (pool) работы, состоящий из сообщений между chare и данных для создания новых chare. Runtime-система (Charm Kernel) может выбирать элементы из этого пула и выполнять их. Она не будет обрабатывать два сообщения для одного chare одновременно, но она свободна отметить их в любое время.

Charm++ обеспечивает динамическую начальную балансировку нагрузки. Таким образом, расположение (номер процессора) не нужно определять при создании удаленного chare. Система сама разместит chare на процессор с наименьшей нагрузкой. Charm Kernel идентифицирует chare по его идентификатору ChareID. Так как пользовательский код не знает, на каком процессоре находится chare, то объект потенциально может мигрировать с одного процессора на другой (это поведение используется для динамической балансировки структуры chare-контейнеров, таких как массивы). Другой вид Charm++-объектов — это chare-контейнеры. Его разновидности — это chare-группы, chare-группы узлов и chare-массивы, соотносящиеся с группами, группами узлов и массивами. Группа — это набор chare, по одному на каждый процессор (SPM-узел), который адресуется использованием уникального для всей системы имени. Массив — это набор произвольного числа мигрирующих chare, отображенных на процессоре согласно определенной пользователем картой групп.

Единственные методы, которые могут вызываться с других процессоров в Charm++, — это асинхронные entry-методы, применяющиеся для удаленного вызова методов объектов chare. Для этого ядро должно знать о типах chare в программе, о методах, которые могут запускаться с удаленных процессоров, их аргументах и т.д. Поэтому при старте программы эти пользовательские объекты нужно зарегистрировать в runtime-системе Charm Kernel, которая при этом назначает каждому из них уникальный идентификатор. При вызове методов на удаленном объекте эти идентификаторы должны быть указаны системе. Регистрация пользовательских объектов и поддержание этих идентификаторов может быть очень громоздким. Поэтому в Charm++ добавлен интерфейсный транслятор. Он генерирует определения прокси-объектов. Кроме того, интерфейсный транслятор позволяет расширять базовую функциональность ядра Charm++ введением потоков и future-объектов пользовательского уровня. Эти потоковые entry-методы могут блокироваться в ожидании данных выполнением синхронного вызова методов удаленного объекта, которые возвращают данные в виде сообщения.

2.5. Распределенные компоненты на основе многопоточности MPI-2. Расширение стандарта MPI-2 [18] развивает эту парадигму в следующих направлениях.

1) *Динамическое создание и уничтожение процессов* (важно для работы в сетях ЭВМ).

2) *Односторонние коммуникации и средства синхронизации для организации взаимодействия процессов через общую память* (для эффективной работы на системах с непосредственным доступом процессоров к памяти других процессоров).

3) *Параллельные операции ввода-вывода* (для эффективного использования существующих возможностей параллельного доступа многих процессоров к различным дисковым устройствам).

4) *C++ интерфейс* (представляет библиотеку MPI в виде нескольких объектов; но он достаточно ограничен для написания объектно-ориентированных программ, использующих все возможности C++).

Введение этих свойств приближает MPI к распределенности и объектной ориентированности.

2.5.1. PAWS. Параллельное пространство приложений (Parallel Application Work Space) PAWS [19] — это программная инфраструктура, используемая для соединения различных параллельных приложений в пределах компонентно-подобной модели. Центральный PAWS-контроллер управляет связями последовательных или параллельных приложений, позволяя им совместно использовать параллельные структуры данных. Приложения используют программный интерфейс PAWS для указания данных, которые могут быть общедоступными, и точек, в которых они готовы для отправления или получения. PAWS реализует общие дескрипторы параллельных данных и автоматически выполняет перераспределение данных, когда это необходимо. Соединения могут динамически устанавливаться и разрываться. Также возможна передача множественных данных между приложениями. PAWS опирается на коммуникационную библиотеку Nexus и не зависит от параллельных коммуникационных механизмов.

Разработчики системы PAWS ставили перед собой следующие цели:

— динамически соединять приложения в любой момент выполнения программы;

— избегать преобразования в последовательную форму критических параметров во время обменов данными между параллельными приложениями;

— разделять данные между параллельными приложениями, используя различные параллельные стратегии размещения;

— обеспечивать обмен данными между приложениями, написанными на разных языках.

Стратегия, которую выбрали разработчики PAWS для решения перечисленных проблем, заключается в реализации стандартного механизма для определения параллельной природы пользовательских типов через программный интерфейс PAWS и в создании PAWS-контроллера (т.е. приложения, управляющего связями между параллельными приложениями и их данными). В этой модели приложения, по сути, становятся компонентами, которые могут связываться через PAWS-контроллер.

В системе PAWS начальное взаимодействие между приложениями происходит через обмен состояниями параллельных структур данных. Приложения могут получать доступ к системе через использование программного интерфейса и присоединение к PAWS-контроллеру. PAWS-контроллер реализован как отдельное приложение. Он похож на простой репозиторий информации о программной среде PAWS. Контроллер содержит в себе следующую информацию: размещение и конфигурация доступных компьютерных ресурсов; список активных приложений, использующих PAWS API; список активных объектов действующих приложений и их характеристики; список установленных соединений между объектами. Контроллер должен быть доступным и работающим при запуске приложений для того, чтобы они могли зарегистрировать себя и свои данные в нем при установлении связи между объектами. Однако при передаче данных между объектами взаимодействие с контроллером не требуется. Информация о состояниях контроллера и рабочей среды хранится в постоянной памяти для улучшения отказоустойчивости системы.

Программный интерфейс PAWS API обеспечивает пользователю возможность добавлять в рабочую среду PAWS новые и существующие параллельные приложения. API реализовано как C++-библиотека. Сначала приложения регистрируют себя в контроллере PAWS. Это осуществляется созданием объекта класса PAWS Application. PAWS использует коммуникационную библиотеку Nexus для связи между разными приложениям и контроллером. PAWS Application после создания устанавливает связь с PAWS-контроллером и со всеми процессами параллельного приложения и обеспечивает пользовательский интерфейс для обращения к контроллеру. Коммуникационный механизм PAWS работает независимо от внутренних способов связи параллельного приложения, таких как MPI, PVM или разделяемая память.

Любой компонент может содержать набор объектов, которые являются общими для нескольких приложений. Все эти объекты должны быть зарегистрированы с помощью PAWS API в контроллере. Для каждого подобного объекта создается экземпляр класса PAWS Data, который хранит в себе состояния соединений к нему. Соединения устанавливаются через запрос к контроллеру или через его скриптовый интерфейс, а также с помощью функций PAWS API.

Модель системы PAWS очень напоминает CORBA (по сути, во многом это реализация идей CORBA на основе коммуникационных механизмов MPI). PAWS можно считать одним из лучших пакетов, появив-

шихся в ходе эволюции стандарта MPI.

3. Промежуточное ПО для распределенных вычислений. Распределенные вычисления появились вместе с технологией удаленного вызова процедур RPC [20]. Эта технология реализует подход распараллеливания по процессам и в первую очередь направлена на эффективное использование вычислительных мощностей многопроцессорных ЭВМ. В основе подхода распределенных вычислений лежат не данные, а процедуры. Распределенная программная система описывает единые данные, для обработки которых запущено множество процессов.

Основной проблемой распределенных вычислений является абстракция данных, т.к. реально в процессе вычислений данные приписаны определенным узлам многопроцессорной ЭВМ. Поэтому большую роль в развитии данного подхода сыграло появление объектно-ориентированного и компонентного подхода. Вслед за RPC появляются компонентные системы, такие как CORBA.

3.1. Интерфейс удаленного вызова процедур RPC. Удаленный вызов процедур (Remote Procedure Call) RPC широко используется в распределенных системах как базовый коммуникационный уровень. В самой общей форме RPC описывает данные, которые должны быть переданы, и удаленные операции, которые должны выполниться с этими данными. Используя простые абстракции удаленного вызова, инициирующий RPC-вызов передается локальному стубу, который маршалирует и передает данные в удаленное адресное пространство посредством стандартных коммуникационных каналов (например, пайпов (pipes), потоков (stream) или TCP). Удаленный стуб демаршалирует данные и передает управление новому потоку (thread), который будет запущен специфической операцией для ассимилирования данных. Результат операции посылается обратно в адресное пространство инициатора вызова через стуб, который затем возобновляет вычисления. RPC-система обычно состоит из IDL-компилятора, генерирующего стубы, и системы реального времени, которая взаимодействует с операционной системой для обработки данных и контроля пересылок.

Сегодня RPC является составной частью многих операционных систем. Данная библиотека позволяет проектировать системы, состоящие из нескольких отдельно откомпилированных программ, взаимодействующих между собой посредством вызовов процедур, реализованных в них. С помощью интерфейса RPC описываются два процесса, участвующих во взаимодействии: один является RPC-клиентом, другой RPC-сервером. Взаимодействия могут быть по типу синхронизации блокируемыми/неблокируемыми, а по количеству одновременных обращений клиентов к серверу — однопоточными/многопоточными. К сожалению, данный стандарт не поддерживает ООП.

3.2. Распределенные компоненты на основе удаленного вызова процедур. Появление объектно-ориентированных языков программирования привело к изменениям в технологиях удаленного вызова процедур: стало необходимо удаленно обращаться не просто к отдельной функции, а к объекту и всем его свойствам и методам. При этом требуется работать с объектами независимо от того, на каком языке они реализованы. Такой подход к объектам стал основой технологий распределенных компонентов.

3.2.1. CORBA. Стандартная архитектура брокера объектных запросов CORBA [21] — это middleware, обеспечивающее взаимодействие распределенных объектов. Распределенные вычисления CORBA основаны на технологии RPC, но ее нельзя понимать как объектно-ориентированную реализацию RPC. Ключевым свойством CORBA является универсальность, т.е. независимость программной системы от языков программирования, каких-либо промежуточных программных средств (операционных систем, сетевых протоколов и пр.) и аппаратных платформ, которая достигается с помощью компонентного подхода. Компоненты, помимо обеспечения объектно-ориентированных распределенных вычислений, позволяют создавать абстрактные объектные модели, которые ранее использовались в объектно-ориентированных языках программирования при создании одной программы. Компонентный подход позволяет расширить границы использования ООП в масштабах распределенных систем, которые объединяют программные компоненты, запущенные как на одном компьютере, так и в сети. Посредством интерфейсов компонентов можно строить различные схемы их взаимодействия, поэтому в CORBA получили развитие клиент-серверные отношения. Основой для компонентного подхода стал язык IDL (Interface Definition Language —- язык описания интерфейсов) [22] — универсальный язык описаний всех компонентов CORBA. Примером эффективного использования этого подхода является инкапсуляция программных приложений, имеющих природу, отличную от CORBA [23]. Это свойство больше известно как интероперабильность (Interoperability). За интерфейсом CORBA можно скрыть компонент, реализованный, например, по технологии MPI. Это позволяет унаследовать большой набор отлаженных, широко используемых программ.

К сожалению, компонентная модель CORBA еще не стандартизована, но уже можно сказать о некоторых ее основных частях. ORB (Object Request Broker — брокер объектных запросов) — компонент, обеспечивающий шину обмена информацией. BOA/POA (Basic/Portable Object Adapter — основ-

ной/переносимый объектный адаптер) — компонент, обеспечивающий процессы создания, использования и уничтожения компонентов CORBA. Общая спецификация объектных сервисов COSS (Common Object Service Specification) — модель компонентов специального назначения, которые обеспечивают общие технические особенности всех компонентов CORBA. Кроме того, составляющими компонентной модели должны стать система реального времени (real-time CORBA) и средства повышения производительности, использующие распараллеливание.

3.2.2. RMI. Технология RMI (Remote Method Invocation — удаленный вызов методов) [24] создавалась одновременно с CORBA и оказала на нее сильное влияние. Это связано с особенностями базового языка программирования Java. Языкозависимость является существенным ограничением (в RMI не используется большое количество программ, написанных на языках, отличных от Java), но в то же самое время является и преимуществом (Java-программы переносятся на различные платформы на уровне байт-кода). В данный момент RMI- и CORBA-компоненты могут взаимодействовать друг с другом. Это стало возможным после внесения существенных изменений в RMI (JDK 1.2) [25] и небольшого пересмотра стандарта CORBA.

3.2.3. COM-технологии. Модель компонентных объектов COM (Component Object Model) [14, 25] — одна из первых компонентных технологий — также имеет все необходимые составляющие: ORB, поддержку статических и динамических вызовов удаленных методов, язык описания интерфейсов, базу данных информации об объектах (библиотеки типов) и др. В основу COM положена двоичная структура объектов, что сразу же обеспечило языконезависимость технологии. С платформонезавимостью хуже: COM-объекты функционируют только на различных версиях ОС Windows.

DCOM (Distributed Component Object Model — модель распределенных компонентных объектов) — расширение COM, описывающее три типа объектов: внутренние (in-proc), локальные (local) и удаленные (remote). На основе DCOM строятся распределенные системы. ActiveX — это еще одна COM-технология, описывающая компонентную модель COM. На основе компонентов ActiveX легко собираются настольные Windows-системы.

В связи с множеством узких мест, COM-технологии могут быть эффективно использованы лишь для создания настольных систем, таких как офис, интегрированная среда разработки программ и др. Применение DCOM ограничивается небольшими автоматизированными системами управления предприятиями и ее использование в распределенных вычислениях кажется сомнительным до тех пор, пока эта технология не будет существенно пересмотрена.

На данный момент существует множество пакетов прикладных программ, которые целесообразно использовать в качестве компонентов распределенных систем; поэтому стандарты мостов COM-CORBA, COM-Java в значительной мере ускоряют процесс разработки сложных систем, позволяя инкапсулировать накопленное ПО. Кроме этого, COM-технологии продолжают бурно развиваться, многие разработки легли в основу новых технологий, возникших в CORBA.

4. Технологии параллельных распределенных компонентов. Эволюция распределенных систем промежуточного ПО идет в нескольких направлениях: организация асинхронных взаимодействий компонентов, реализация MPI-подобных интерфейсов, создание низкоуровневых систем middleware и реализация интерфейсов для параллельных распределенных вычислений, интеграция и инкапсуляция параллельных интерфейсов в распределенные системы. Далее рассмотрим известные проекты, появившиеся в этой области.

4.1. Параллельные распределенные компоненты (COM+, CORBA 3.0). Направление стандартизации высокопроизводительных вычислений в COM и CORBA связано с развитием клиент-серверных отношений, сервисов синхронных/асинхронных сообщений, сервисов качества (изменяющих поведение коммуникационной инфраструктуры с целью достижения максимальной эффективности). Если в COM+ уже существуют подобные решения, то в стандарте CORBA 3.0 они только планируются. На новый стандарт, по-видимому, большое влияние окажут COM+ (Windows 2000) и попытки реализовать вышеупомянутые сервисы в некоторых коммерческих системах CORBA. В новых стандартах распределенных систем появляются следующие основные нововведения.

1) *Многопоточность серверов.* Это позволяет более эффективно обрабатывать запросы клиентов, которые обращаются к компоненту (серверу) с вызовом того или иного метода. Предлагаются различные способы программирования серверной части: выделение отдельных потоков для каждой клиентской сессии, подготовка пула (накопителя) потоков и др.

2) *Асинхронные взаимодействия на основе интерфейсов MSMQ* (Microsoft Message Queuing — очередь сообщений COM+) *и AMI* (Asynchronous Method Invocation — асинхронный вызов методов CORBA). Данные интерфейсы описывают асинхронные вызовы и их обработчики.

3) *Средства автоматической оптимизации, названные QoS* (Quality of Services — служба качества сервисов COM+ и CORBA). Новые системные компоненты реализуют алгоритмы по управлению компонентами разной природы (транзакционных, асинхронных, многопоточных) относительно аппаратной среды: ее архитектуры и коммуникационных возможностей.

4.2. Построение параллельных распределенных систем на основе Java-платформ (EJB, ParJava). Существует ряд параллельных распределенных Java-технологий. Остановимся на EJB (Enterprise Java Beans) [22, 24] и ParJava [26]. Общее в этих системах то, что они построены уже на основе промежуточного ПО Java, реализуя коммуникационную среду для компонентов. Отличие заключается в том, что EJB основывается на стандарте CORBA и распределенных вычислениях, а в ParJava главной целью является описание SIMD-программ, для чего предлагаются объектно-ориентированный интерфейс MPI и собственный интерфейс, описывающий структуру ВС.

В основе EJB лежат технологии RMI и CORBA. Любой компонент bean является компонентом CORBA; отсюда следуют все возможности по работе с bean. По сути, в EJB реализованы на языке Java интерфейсы CORBA, в частности интерфейсы для коммуникации компонентов. Это позволяет строить распределенные приложения и использовать средства распараллеливания CORBA.

ParJava — это технология SIMD-программирования на Java. В качестве коммуникационной среды выбран стандарт MPI. С помощью JNI (Java Native Interface — интерфейс вызова стандартных функций C++ из Java) инкапсулирована MPI-система LAM. Поверх интерфейса коммуникационной среды реализована библиотека масштабирования Java-приложений на однородные и неоднородные сети. В этой технологии используются JIT-компиляторы (Just-In-Time — компиляторы реального времени), увеличивающие производительность виртуальных машин JavaVM.

4.3. Вычислительные сети GLOBUS. Международный проект Globus представляет технологию разработки вычислительных сетей (Grid — сеть) [27]. Под вычислительной сетью понимается программно-аппаратная среда, позволяющая интегрировать аппаратные, вычислительные и информационные ресурсы, которые принадлежат различным организациям и распределены по Internet. Основной особенностью Globus является четкое определение архитектуры ВС. Grid-архитектура определяет фундаментальную систему компонентов. Можно отметить следующие особенности этой архитектуры.

1) В первую очередь данная архитектура является *протоколом*, по которому пользователи взаимодействуют с ресурсами, что обеспечивает интероперабильность (возможность функционирования приложения в гетерогенной аппаратно-программной среде).

2) Ряд стандартных возможностей системы определяется специальными компонентами — *сервисами*, которые определяют поведение компонентов системы.

3) Важными составляющими архитектуры являются API (Application Programming Interface — программный интерфейс) и SDK (Software Development Kit — библиотека разработчика). Стандартные абстракции (API и SDK) увеличивают скорость разработки, позволяют повторно использовать программный код, улучшают переносимость приложений.

Реализован пакет Globus как самостоятельная система, на базе которой переписан интерфейс MPI для параллельных вычислений и интегрированы компонентные системы CORBA, EJB. В отличие от других технологий распределенных вычислений Globus сразу же был ориентирован на высокопроизводительные вычисления и широкую масштабированность.

4.4. Инкапсуляция и интеграция MPI в CORBA. Одним из наиболее перспективных направлений с точки зрения теоретического программирования является развитие общих моделей SIMD. Данный подход позволяет отразить в IDL-описании компонента его возможное поведение либо путем использования специальных интерфейсов [28], либо путем расширения синтаксиса IDL [29]. Это позволяет значительно упростить разработку параллельных программ еще и потому, что описания интерфейсов CORBA могут быть использованы специально вводимыми компонентами (сервисами), отрабатывающими различные методы распараллеливания. Реализовать SIMD-технологии можно путем использования промежуточного ПО MPI в составе CORBA.

4.4.1. PARDIS. Программная система PARDIS [30], являющаяся примером инкапсуляции MPI в CORBA путем использования специальных интерфейсов и сервисов, состоит из компилятора IDL, коммуникационных библиотек, базы данных объектного репозитория и служб, ответственных за размещение и активацию объектов. Перечислим основные компоненты PARDIS.

1) *Параллельные серверы*: программы, которые обеспечивают выполнение одиночных и SIMD-объектов. Параллельный сервер – это набор из одного или более вычислительных потоков, определенных во время запуска сервера или во время активизирования одного из его параметров. Предполагается, что эти потоки используют некоторую коммуникационную среду, отличную от коммуникационных библиотек PARDIS

(например, библиотеку передачи сообщений), для связи определенных программ. Текущая реализация предполагает, что потоки соотносятся с распределенной моделью памяти.

2) *Параллельные клиенты*: программы, составленные из одного (или более) вычислительного потока и способные запрашивать сервисы у параллельных объектов, действуя как одна сущность (SIMD-объект) или как отдельный объект. Как и в случае с серверами, принимается, что клиенты могут общаться, используя отличные от PARDIS коммуникационные библиотеки.

3) *ORB* (Object Request Broker): сущность, отвечающая за выполнение запросов между клиентами и серверами. Чтобы должным образом обрабатывать запросы, объектный брокер ORB может нуждаться в связи с исполнительной системой, лежащей в основе параллельного сервера или клиента. Для доступа клиента к реализации объекта программист должен реализовать объект в терминах выбранного пакета и определить его интерфейс на IDL. Компилятор IDL транслирует описание объекта в код стуба, содержащего вызов ORB. Связанный с описанием объекта стуб позволяет объектному брокеру вызывать его методы. Клиент может использовать стубы для отправки запросов на сервер.

4) *Интерфейс исполнительной системы*, через которую брокер взаимодействует с клиентами и серверами, включает в себя вызовы базовых функций коммуникации и маршалинга данных, специфичных для данной системы. Функциональные требования ограничены небольшим подмножеством основных функций передач сообщений. Чтобы избежать конфликтов, система требует различать сообщения PARDIS от сообщений, принадлежащих вычислениям в пользовательском коде (например, через множество зарезервированных пометок (tags) сообщений). Пока этот интерфейс реализован на MPI, системе TULIP и коммуникационных абстракциях библиотеки POOMA, что позволяет PARDIS взаимодействовать с объектно-ориентированными пакетами, построенными на базе этих систем. Ограничение исполнительной системы малыми установленными пределами функциональности распределенных структур позволяет обеспечивать взаимодействие с большим числом параллельных пакетов.

5) *Репозитории объектов и реализаций*: база данных, определяющая именное пространство взаимодействующих объектов. При активации каждый объект регистрируется в объектном репозитории, вызываемом, когда клиент посылает запрос к нему. Каждый репозиторий ассоциируется с уникальным именным пространством; настройка клиентов и серверов для работы с разными репозиториями позволяет программисту разделять именные пространства взаимодействующих объектов. В случае непостоянных серверов пользователь может использовать специальные средства для регистрации объекта и информации о том, как этот объект будет активироваться из репозитория реализаций.

6) *Формирование средств*: с того времени как установлено соединение с объектом, для запуска реализующего его сервера PARDIS обеспечивает активизацию агентов. Агенты обычно находятся на главном компьютере сервера. Для ограничения взаимодействия между активным агентом и сервером программист может настроить систему для работы в активном или неактивном режимах.

Объекты создаются серверными программами. SIMD и одиночные объекты могут разделять ресурсы одного и того же параллельного сервера. При инсталляции объекта программист должен определить его тип (одиночный или SIMD-объект). Размещение SIMD-объекта осуществляется коллективно относительно всех вычислительных потоков сервера. Одиночный объект может быть создан только как объект, не имеющий распределенных параметров.

Для взаимодействия с объектами клиент должен установить соединение между локальным стубом объекта и его реализацией. Клиент может соединить объекты двумя способами: или коллективно (брокер представляет параллельного клиента как одну сущность), или вызовом связывания, создающего одно соединение на поток. Все методы на локальном стубе объекта, вызванные использованием коллективного (SIMD) связывания, начинают работать одновременно и могут использовать распределенные аргументы. Для поддержки вызовов операций одиночного объекта, имеющих распределенные аргументы, PARDIS генерирует два вида стубов для каждого метода: один с распределенным расположением для поддержки SIMD-вызовов, а другой — с передачей нераспределенных аргументов для одиночных вызовов.

Чтобы полностью использовать взаимодействие с SIMD-объектами, необходимо определять и манипулировать аргументными структурами данных, распределенных в адресных пространствах потоков SIMD-объекта. PARDIS поддерживает одну такую структуру, названную распределенной последовательностью. Она является обобщением CORBA-последовательности. IDL определение

```
typedef dsequence!double, 1024, (BLOCK,ONE)? Dist*seq*double;
```

представляет ограниченную распределенную последовательность из 1024 элементов типа double, равномерно и блочно распределенную на клиентской стороне, но сконцентрированную на одном процессоре серверной стороны. Последние два аргумента определяют опции последовательности. Основываясь на этом определении IDL-компилятор генерирует C++-класс, который ведет себя как одномерный массив

с переменной длиной и распределением. Если никакого распределения не было указано, то можно установить его, используя шаблон распределения. Описывается пропорция распределения элементов последовательности среди процессоров. Используя различные шаблоны, достаточно перераспределять последовательность. Сервер способен устанавливать распределение любых входных элементов своих методов до регистрации объекта. Клиент может определить распределение выходных параметров до начала выполнения метода. Знание распределения аргументов позволяет объектному брокеру эффективно пересылать аргументы между клиентом и сервером.

Для поддержки параллелизма PARDIS использует как блокируемый, так и не блокируемый вызов методов на сервере. Для каждого метода IDL-компилятор генерирует два стуба. Операция, вызванная через блоковый стуб, возвращает управление только после того, как полностью отработает сервер. Метод же, запущенный неблокируемым стубом, возвращает управление немедленно, после того как запрос был послан, с будущими (future) выходными параметрами и возвращаемым значением. Здесь слово "будущие" означает, что результат выполнения операции может быть еще не доступен. Попытка прочитать будущие (еще не возвращенные) данные приводит к блокированию программы до того момента, пока они не будут получены. Другой способ — проверка, когда эти данные придут. Неблокируемые вызовы могут использовать множественные будущие данные распределенно. Они все могут быть определены в то время, когда сервер завершит вычисления.

4.4.2. Cobra. Одним из наиболее удачных примеров интеграции стандартов CORBA и MPI является проект Cobra [31]. Разработчики Cobra применили технологию, позволяющую MPI-коду быть инкапсулированным в компонент. Основная цель проекта — это обеспечение программной инфраструктуры для приложений распределенного моделирования на вычислительных сетях. Эта технология базируется на расширении OMG CORBA, в частности одной из ее разновидности MICO. Предложенные расширения не модифицируют инфраструктуру ядра CORBA (ORB — Object Request Broker —- брокера объектных запросов) до такой степени, что оно не может использоваться с другими уже существующими CORBA-приложениями. MPI-код определяется как новый вид CORBA-объекта, который скрывает большую часть проблем, связанных с параллелизмом.

В системе Cobra показано, как можно расширить CORBA для поддержки параллельности в компонентах, использующих SIMD-модель выполнения. Точнее, предложенная технология позволяет инкапсулировать MPI-код в новый вид CORBA-объекта, названного параллельным CORBA-объектом. Параллельный CORBA-объект — это просто коллекция идентичных CORBA-объектов. Они могут управляться как единый объект системы. Основной целью является сокрытие, насколько это возможно, проблем, связанных с распределенной памятью на параллельных компьютерах, например на кластерах. Прежде всего, вызов клиентом некоторой функции выполняется всеми объектами коллекции, поддерживающими SIMD-модель запуска. Это параллельное действие производится системой Cobra. Распределение данных также полностью поддерживается системой. Для выполнения параллельных операций и распределения данных вводятся расширения спецификации объектного интерфейса. В этой системе расширенный язык описания интерфейса называется ExtendedIDL. В IDL-синтаксис добавлены несколько ключевых слов для представления обоих типов объектов: распределенные данные и коллекции.

Существуют две причины добавления MPI в CORBA. Основная — это позволить параллельному объекту инкапсулировать MPI-код. Такие коды должны основываться на SIMD-модели, так как MPI-процесс соответствует реализации объекта. Поэтому необходимо предложить интерфейс передачи сообщений объекта. Вторая причина — это потребность в использовании коммуникационного уровня для распределения и перераспределения данных между объектами в коллекции. Стубы, сгенерированные ExtendedIDL-компилятором, имеют функции для обмена данными и синхронизации между собой. Перед тем как запустить на выполнение операцию, имеющую распределенный параметр, должны быть собраны данные из объектов коллекции. Кроме того, только один объект коллекции должен быть выбран для запуска на серверной стороне. Остальные должны ждать, пока не завершится операция. Подобные проблемы возникают, когда один параллельный объект вызывает метод другого параллельного объекта. Если существует распределенный параметр, данные должны быть перераспределены согласно как с распределением данных вызывающего объекта, так и с распределением данных у вызываемого объекта. Подобные операции обрабатываются удаленным стубом вызываемого объекта, который знает распределение данных, и стубом клиента.

И CORBA, и MPI предлагают свои службы удаленного запуска, когда запускается MPI-код или вызывается удаленный объект. Такие службы часто зависят от конкретной реализации CORBA и MPI. В этой системе выбраны MICO и MPICH. Обе эти среды существуют на большом количестве платформ. В CORBA запуск сервера осуществляется через Basic Object Adapter (BOA), который отвечает за загрузку

и выполнение процессов, реализованных в объекте, при их вызове клиентом. Информацию о соответствующей реализации BOA получает из репозитория реализаций. В данном случае запуск параллельных объектов не может быть представлен стандартным BOA. Использование функциональности исполнительной (runtime) системы MPI вынуждено. Предложено интегрировать ее в BOA, который был назван Parallel BOA (PBOA). С помощью PBOA запуск параллельного объекта осуществляется в два действия. В начале активируется один из объектов коллекции. Он создает коммуникационную группу MPI, которая понадобится объектам коллекции для синхронизации и обменов данными. Затем он активирует остальные объекты коллекции, предоставляя им коммуникационную группу, к которой они должны будут присоединиться. Система Cobra предоставляет узлы расположения этих объектов посредством нескольких CORBA-служб размещения ресурсов.

Вызов параллельного объекта осуществляется следующим образом. Сначала клиент определяет группу узлов для своих вычислений, использующих Cobra, вызовом специальной функции. Затем он просит каждого PBOA (UNIX процесс, запущенный на каждом узле группы) активизировать свои службы. Каждый PBOA посылает запрос к Cobra для того, чтобы знать, какой PBOA отвечает за запуск функции mpirun MPICH. Эта команда запускает MPI-процессы на каждом узле группы. Когда объекты готовы, они возвращают ссылку на себя PBOA. Каждая ссылка затем отправляется клиенту. Все эти шаги скрыты от клиента.

Разработчики Cobra не стали добавлять к системе CORBA новых служб для того, чтобы обрабатывать действия связанные с новыми типами объектов (параллельные, распределенные объекты). Вместо этого они частично переписали уже существующие службы CORBA.

5. Требования к промежуточному ПО для высокопроизводительных вычислений. Анализ существующего промежуточного программного обеспечения позволяет выделить ряд основных свойств технологий высокопроизводительных вычислений. Возможно, этот список неполон. Несомненно, он будет расширяться в связи с развитием вычислительной техники и технологий программирования.

5.1. Языконезависимость. Языконезависимость можно считать основным признаком технологии программирования, когда прикладная программа может быть написана на любом, удобном для разработчика языке. Во всех системах языконезависимость достигается двумя способами: путем реализации интерфейса системы на различных языках программирования и путем введения специальных высокоуровневых описаний, которые затем транслируются на различные языки.

1) Реализация интерфейса системы на различных языках программирования. Такой подход применяется, например, в технологии MPI. Интерфейс MPI реализовывается только на языках C и Fortran. Недостаток такого подхода — несовершенный интерфейс. В связи с развитием языков программирования, в частности с появлением объектно-ориентированных языков, возникла необходимость записать интерфейс MPI совершенно иным способом. Поэтому сначала появляются интерфейсы MPI++ и OOMPI, в которых сама промежуточная среда представляется в виде объектов, затем создаются описания параллельных объектов TPO++ и интерфейсы для работы с компонентами.

2) Введение специальных высокоуровневых описаний. В технологии CORBA центральное место занимает язык описания интерфейсов IDL, который эволюционирует вместе с самой системой. Для того чтобы расширить языковую поддержку CORBA, необходимо реализовать транслятор, переводящий IDL-описания в текст на целевом языке. IDL-описания используются не только для получения исходного кода программы, но также и для автоматизации управления компонентами. Кроме этого, выделение языка как части системы позволяет CORBA не отставать от развития языков программирования.

5.2. Платформонезависимость. Платформонезависимость также можно считать основным признаком технологии вычислений. Как показал анализ, независимость от компьютерной платформы может достигаться на уровнях исходного кода, промежуточной среды, бинарного кода.

1) Переносимость на уровне исходного кода: в данном случае все прикладные программы переносимы, но для этого требуется их перекомпиляция на новой платформе. Взаимодействовать друг с другом компоненты разных платформ не могут. Перенос обеспечивается единым интерфейсом на общем языке программирования. Процесс усложняется или вообще невозможен, если используются библиотеки, связанные с той или иной платформой. MPI — пример системы middleware, платформонезависимой на уровне исходного кода.

2) Переносимость на уровне промежуточной среды: переносимость на уровне исходного кода и, кроме того, программы, работающие на разных платформах, могут взаимодействовать друг с другом. Функционирование многокомпонентной системы обеспечивается единым интерфейсом, причем использование общего языка программирования не обязательно. В качестве примеров можно привести надстройку над различными системами MPI для разных платформ, реализующую общую промежуточную среду, и тех-

нологию CORBA.

3) Переносимость на уровне бинарного кода: переносимы откомпилированные прикладные программы. Промежуточная система, реализующая такой подход, включает в себя некую виртуальную машину для выполнения программ. Такого рода перенос возможен между различными версиями ОС Windows посредством языконезависимой технологии COM и между ОС, оснащенными Java-системой. В данном случае узким местом является виртуальная машина: необходимо, чтобы она была реализована для широкого класса ОС, была достаточно производительной и использовала наработанное ПО для данной ОС.

Наиболее приемлемыми на данный момент являются системы, платформонезависимые во втором смысле, такие как CORBA. Их преимущество — эффективное взаимодействие с платформой, т.к. доступ к ней осуществляется без посредников, таких, как виртуальная машина JavaVM. Платформа понимается широко: операционная система, созданное на ее основе программное обеспечение, аппаратные средства. В данном случае может оказаться полезнее некоторая зависимость от платформы, чем полный отказ от нее, как это сделано в JavaVM. С другой стороны, очевидной становится проблема управления программными компонентами в гетерогенной аппаратно-программной среде. Чтобы решить эту проблему, можно расширить функциональность CORBA Application Server путем внедрения компиляторов, интерпретаторов и библиотек к ним, что позволит управлять исходным кодом компонентов.

5.3. Адаптация к различным архитектурам аппаратных средств. Промежуточное программное обеспечение позволяет абстрагироваться от аппаратных средств, поэтому его введение чревато неэффективным использованием компьютерных ресурсов. Важно, чтобы на первом месте стоял не стандартный программный интерфейс, а максимальная производительность.

MPI включает описания процессоров, их различных топологий, оптимизирует вычисления для виртуальных топологий, задаваемых программно. Это позволяет более эффективно отобразить прикладную задачу на сеть вычислительных узлов. В ParJava реализован интерфейс, похожий на MPI, посредством которого можно эффективно использовать узлы с виртуальной машиной Java в параллельных вычислениях.

В последнее время в распределенные системы включаются элементы, позволяющие учитывать архитектуру ЭВМ. В некоторые сервера приложений CORBA и в Windows 2000 Server включены сервисы QoS, повышающие качество взаимодействий между распределенными объектами в зависимости от возможностей коммуникационной среды. Кроме этого, в компонентные технологии включаются системы реального времени, позволяющие эффективно управлять компонентами, находящимися на разных узлах.

Анализ показал, что самыми перспективными являются технологии с гибкими расширяющимися интерфейсами. Это позволяет не только адаптировать многокомпонентные системы к возможностям ЭВМ, но и создавать программные средства для этого (например, автоматическая балансировка нагрузки в COM+). На базе Windows 2000 Advanced/Super Server функционируют системы балансировки нагрузки CLB и NLB. Подобные сервисы появляются в составе QoS в некоторых реализациях CORBA.

5.4. Использование различных моделей распараллеливания. Широкий набор описаний процессов и данных позволяет точно записать задачу в терминах данной технологии. Многие большие задачи подразумевают массивный параллелизм и требуют значительных вычислительных ресурсов. От того, насколько правильно описаны параллельные процессы и распределенные данные, будет зависеть производительность ВС.

Эволюция параллельных и распределенных систем (несмотря на то, что они использовались для решения определенных классов задач) показывает, что сегодня высокопроизводительная технология должна описывать и SIMD-, и MISD-, и MIMD-модели. Наметилась тенденция к расширению описательных возможностей существующих стандартов (для MPI разрабатываются технологии распределенных объектов, для CORBA — дополнительные модули параллельной обработки). Такие системы должны строиться на основе компонентного подхода с широким использованием различных методов взаимодействия между процессами (обмены) и доступа к объектам (клиент-сервер).

5.5. Объектно-ориентированный подход. Объектно-ориентированный стиль моделирования и программирования систем на настоящий момент времени является наиболее удобным и эффективным для реализации программных (и не только программных) систем. Объектно-ориентированный подход (ООП) [32] основан на систематическом использовании моделей для языковонезависимой разработки программной системы.

Модель содержит не все признаки и свойства представляемого ею предмета, а только те, которые существенны для разрабатываемой программной системы. Таким образом, модель есть формальная конструкция: формальный характер моделей позволяет определить формальные зависимости между ними и формальные операции над ними. Это упрощает как разработку и анализ моделей, так и их реализацию

на компьютере. В частности, формальный характер моделей позволяет получить формальную модель разрабатываемой программной системы как композицию формальных моделей ее компонентов. Поэтому объектно-ориентированный подход помогает справиться с такими сложными проблемами, как:

1) уменьшение сложности программного обеспечения;

2) повышение надежности программного обеспечения;

3) модификация отдельных компонентов программного обеспечения без изменения остальных его компонентов;

4) повторное использование отдельных компонентов программного обеспечения.

Систематическое применение объектно-ориентированного подхода позволяет разрабатывать хорошо структурированные, надежные в эксплуатации, достаточно просто модифицируемые программные системы. Этим объясняется интерес программистов к объектно-ориентированному подходу и объектно-ориентированным языкам программирования. Объектно-ориентированный подход является одним из наиболее интенсивно развивающихся направлений теоретического и прикладного программирования.

ООП оказал существенное влияние как на интерфейс, так и на структуру ВС. Эволюция параллельных и распределенных вычислений привела к оформлению объектной структуры высокопроизводительной системы, к описанию обобщенных объектных интерфейсов.

5.6. Компонентный подход. На основе ООП развивается компонентный подход, в котором этапы проектирования и программирования четко разграничены. Компонентом называют объект, в котором разделены интерфейс и реализация. Это позволяет реализовать ее одновременно на разных платформах. Прикладное обеспечение, созданное на основе такой системы, может функционировать на многих платформах. Как компонентная технология с самого начала проектировалась CORBA. Показательно, что для системы MPI в ходе эволюции были определены объектная структура, интерфейс работы с объектами, компонентная модель. Компонентный подход позволяет выделить ряд процессов разработки программ, которые могут быть автоматизированы, поэтому важно создать соответствующие средства разработки. В настоящее время наметилась тенденция к созданию интегрированных сред разработки, направленных не только на создание программ на каком-либо языке программирования, но и поддерживающих различные технологии программирования. Так, например, Microsoft Visual Studio позволяет создавать объектно-ориентированные программ на языках C++, Visual Basic, J++ с поддержкой COM-технологий, а Borland Enterprise Studio поддерживает Java и CORBA. Можно выделить ряд необходимых свойств среды разработки:

1) наличие инструментов объектного моделирования (например, Rational Rose);

2) поддержка компонентной технологии (например, CORBA);

3) тесная интеграция всех инструментов для повышения эффективности программирования семантики компонентов;

4) возможность групповой работы над проектами, для чего создаются CVS-системы (Control Version System — система контроля версий);

5) ведение библиотек компонентов для их повторного использования (например, репозитории CORBA);

6) наличие хранилища шаблонов и мастеров для автоматизации создания компонентов;

7) открытость, т.е. возможность включения в среду разработки новых компонентов (компиляторов, скриптов, редакторов и др.).

Среду разработки, удовлетворяющую данным требованиям, имеет смысл создавать на основе существующих инструментов, таких как Microsoft Visual C++, Rational Rose, некоторых компонентов CORBA (IDL-компилятор, репозитории объектов, сервис поддержки жизненного цикла и др.), языков управления сценариями Perl, Shell, Tcl.

5.7. Инкапсуляция и интеграция существующих ВС. Функциональные возможности и скорость разработки прикладных программных систем напрямую зависят от технологий, которые лежат в их основе. Если технологии имеют эффективные механизмы повторного использования программ (объектно-ориентированный и компонентный подходы), то появляется возможность инкапсуляции и интеграции существующих ВС.

1) Инкапсуляция — это включение в состав многокомпонентной системы набора компонентов некоторой ВС. Это позволяет повторно использовать программные компоненты. В тех случаях, когда ВС, которую необходимо инкапсулировать, основана на другой технологии, создаются специальные механизмы, например, программы-мосты между компонентными системами COM-CORBA, Java-COM, CORBA-RMI. Инкапсуляцию существующих ВС может осуществить прикладной программист.

2) Интеграцией называется инкапсуляция с одновременным изменением архитектуры исходной системы. Интеграцией MPI в CORBA можно назвать проект Cobra, когда модифицируются важные соста-

вляющие CORBA. Это относится к области системного программирования. Перед тем как браться за интеграцию, необходимо оценить преимущества и недостатки видоизменения базовой ВС.

Создание ВС является комплексной задачей, для решения которой необходимо решить ряд проблем, не связанных с основными вычислительными целями. Поэтому для увеличения скорости разработки ВС полезно инкапсулировать существующие системы визуализации данных, CAD-системы, некоторые стандартные математические библиотеки.

Интеграция в ВС каких-либо систем имеет смысл, когда на базе этих систем разработаны необходимые библиотеки. Интеграция полезна также при появлении в базовой ВС новых моделей распараллеливания. Сборка интегрированной системы является эффективным способом создания проблемно-ориентированных сред, объединяющих вычислительные системы различных дисциплин в единый программный комплекс [33, 34].

5.8. Система управления. Рассмотрение компонентных систем показало значение компонентной модели и системы реального времени для управления программными компонентами. Мало спроектировать и запрограммировать многокомпонентную систему, необходимо ее установить и отследить во время выполнения. Если процесс разработки компонента определяется описанными выше характеристиками ВС, то что можно сказать о процессе его внедрения и использования? Этот процесс нужно и можно формализовать.

Для описания эксплуатации компонентов необходимо определить структуру компонента, которая задается компонентной моделью, и основные действия над ним, которые будут выполняться системой реального времени. Реализациями такого подхода являются Windows 2000 Server и Borland Application Server, использующие следующие приемы.

1) Определяется интерфейс компонента, включающий наиболее основные и общие свойства. Этот интерфейс по сути задает класс компонентов, например, транзакционные компоненты COM+.

2) Определяется интерфейс сервиса, обслуживающего компоненты данного класса. Сюда включаются основные методы управления компонентами: отслеживание состояния, активация, деактивация и т.д.

3) Сервис реализуется и включается в компонентную систему. Создается средство интерактивного взаимодействия с сервисом.

Средства управления играют самую значительную роль в процессе использования компонентов, позволяя конфигурировать их в соответствии с архитектурой аппаратной среды. На сегодня сделаны попытки создания технологий масштабирования распределенных систем, балансировки нагрузки, они включены, например, в COM+ и в Borland Application Server.

Таким образом, в ходе рассмотрения существующих на данный момент систем выделены необходимые характеристики высокопроизводительной ВС и определены основные технические стандарты, на основе которых она может быть построена. Подобная система разрабатывается в рамках проекта, направленного на создание промежуточного ПО для высокопроизводительных вычислений.

СПИСОК ЛИТЕРАТУРЫ

1. *Уемов А.И.* Системный подход и общая теория систем. М.: Мысль, 1978.
2. *Ортега Дж.* Введение в параллельные и векторные методы решения линейных систем. М.: Мир, 1991.
3. *Dongarra J.J. and et al.* PVM version 3.4: Parallel Virtual Machine system. University of Tennessee (Knoxville TN), Oak Ridge National Laboratory (Oak Ridge TN), Emory University (Atlanta GA), 1997.
4. *M*PI Forum. MPI: A Message-Passing Interface standard. 1998 (http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html).
5. *Воеводин Вл.В.* Технологии параллельного программирования. Message Passing Interface (http://parallel.srcc.msu.su/vvv/mpi.html).
6. *M*PICH-G2 (http://www.niu.edu/mpi/).
7. *T*he Globus project (http://www.globus.org/).
8.. *T*he Globus project. The Nexus Multithreaded Communication Library (http://www.globus.org/nexus/).
9. *Bangalore P.V., Doss N.E., Skjellum A.* MPI++: issues and features // OON-SKI'94. 1994. 323–338.
10. *Squyres J.M., McCandless B.C., Lumsdaine A.* Object-Oriented MPI (OOMPI): A C++ class library for MPI. Version 1.0.2f (http://www.lsc.nd.edu/research/oompi/documentation.htm).
11. *Skjellum A., Lumsdaine A., Bangalore P. et al.* Object-Oriented MPI design and implementation // Concurrency: Practice and Experience. 2001 (to appear).
12. *Grundmann T., Ritt M., Rosenstiel W.* TPO++: An object-oriented message-passing library in C++ // Proc. Lilja-2000. 2000. 43–50.
13. *Kale L.V., Ramkumar B., Sinha A., Gursoy A.* The Charm parallel programming language and system. Part I — description of language features. Technical Report 95–2. Parallel Programming Laboratory, Department of Computer

Science, University of Illinois, Urbana–Champaign, 1995.
14. *Rogerson D.* Inside COM. Redmond (Washington): Microsoft Press, 1996.
15. *Myers N.C.* Traits: a new and useful template technique // C++ Report. Chatsworth (California): SIGS Publications. 1995. **17**, N 5. 32–35.
16. *McKenna F. T.* Object-oriented finite element programming: frameworks for analysis, algorithms and parallel computing. PhD thesis (University of California). Berkeley, 1997.
17. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. М.: Бином, 1999.
18. *M*PI Forum. MPI-2: extensions to the Message-Passing Interface. 1998 (http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html).
19. *Beckman P.H., Fasel P.K., Humphrey W.F.* Efficient coupling of parallel applications using PAWS // Proc. of the Seventh IEEE Symposium on High-Performance Distributed Computing (HPDC-7). New Brunswick (New Jersey), 1998. 215–222.
20. *Birrel A., Nelson G.* Implementing remote procedure calls // ACM Transactions on Computer Systems (TOCS). 1984. **2**, N 1. 39–59.
21. *O*bject Management Group. The Common Object Request Broker: architecture and specification (Revision 2.3.1). 1999 (http://www.omg.org/corba).
22. *Цимбал А.* Технология CORBA. Для профессионалов. СПб.: Питер, 2001.
23. *Vinoski S.* CORBA: integrating diverse applications within distributed heterogeneous environment // IEEE Communication Magazine. 1997. **14**, N 2. 46–55.
24. *T*he source for Java ™ Technology (http://java.sun.com).
25. *Brockschmidt K.* Inside OLE. Redmond (Washington): Microsoft Press, 1993.
26. *Аветисян А.И., Арапов И.В., Гайсарян С.С., Падарян В.А.* Параллельное программирование с распределением по данным в системе ParJava // Вычислительные методы и программирование. 2001. **2**, № 1. 129–146.
27. *Foster I., Kesselman C., Tuecke S.* The anatomy of the grid. Enabling scalable virtual organizations // The International Journal of Supercomputer Application. 2001. **15**, N 3 (to appear).
28. *Keahey K., Gannon D.* Collective Objects: an object-oriented tool for collective operations in distributed parallel computation. Report TR 461. Department of Computer Science, Indiana University. Indiana, 1996.
29. *Rene C., Priol T.* MPI code encapsulating using parallel CORBA object // Proc. of the Eighth IEEE International Symposium on High Performance Distributed Computing. New Brunswick (New Jersey), 1999. 3–10.
30. *Keahey K., Gannon D.* Pardis: a parallel approach to CORBA // Proc. of the Sixth IEEE International Symposium on High Performance Distributed Computing. New Brunswick (New Jersey), 1997. 31–39.
31. *Priol T., Rene C.* Cobra: A CORBA-compliant programming environment for high-performance computing // Proc. of Euro-Par'98 (UK). Southampton: Springer Verlag, 1998. 1114–1122.
32. *Гайсарян С.С.* Объектно-ориентированные технологии проектирования прикладных программных систем (http://www.citforum.ru/programming/oop_rsis/index.shtml).
33. *Walker D.W.* The software architecture of a distributed problem-solving environment // Concurrency — Practice and Experience. 2000. **12**, N 15. 1455–1480.
34. *Буньков Н.Г.* Распределенные интерактивные вычисления в реализации многодисциплинарных проблем расчета летательного аппарата // Новое в численном моделировании: алгоритмы, вычислительные эксперименты, результаты. М.: Наука, 2000. 228–246.

Поступила в редакцию
02.11.2001