Методика распределенных вычислений RiDE

М.О. Бахтерев, П.А. Васёв, А.Ю. Казанцев, И.А. Альбрехт

ИММ УрО РАН

RiDE - это методика для программирования в параллельных распределенных средах, основанная на модели потока данных (dataflow). RiDE превосходит подходы MPI и OpenMP по простоте использования. Также RiDE превосходит более узкие парадигмы, например Map/Reduce и Task Flow, позволяя решать более широкий класс задач. Методика обладает такими преимуществами, как автоматическая балансировка процесса вычисления, распространение счета при добавлении/удалении узлов "на лету", полная остановка счета и освобождение ресурсов с дальнейшим возобновлением, автоматическое создание контрольных точек, работа в гетерогенных и гибридных (CPU/GPU/CELL/ПЛИС/etc) средах.

1. Введение

Распространенные средства параллельного программирования, такие как MPI [1], OpenMP [2] и GLOBUS [3], требуют от программиста подробного описания большого количества сущностей. Необходимо заботиться о распределении вычислительных задач, синхронизации, обмене данными и так далее. В связи с этим, программирование с использованием этих средств является трудоемкой задачей, отвлекающей на себя существенную часть рабочего времени программиста-математика. Программы создаются долго, получаемые коды сложны и громоздки, их сопровождение и развитие оказывается затратным процессом.

Другой проблемой распространенных средств параллельного программирования является их ориентация на конкретные классы вычислительных систем: с общей памятью, кластерные системы либо распределенные системы. Программы, написанные с помощью таких средств, способны выполняться на подразумевавшемся типе параллельной системы, но не способны эффективно работать с системой другого типа. Кроме того, данные средства, как правило, не содержат встроенной поддержки ускорителей (GPU, ПЛИС и т.д.), а также концепций облачных вычислений и SaaS.

Существуют различные подходы к упрощению процесса программирования и исполнения параллельных вычислений. С одной стороны, создаются универсальные средства по автоматизации распараллеливания программ (как для исполнения в системах с общей памятью, так и в многомашинных конфигурациях, например DVM [4] и Т-Система[5]). С другой стороны, создаются среды для решения определенных классов задач (в основном это касается задач, для которых применим параллелизм «по данным», например Map/Reduce [6]). Также разрабатываются универсальные инструменты, пытающиеся упростить технические аспекты процесса программирования (например Intel TBB [7]).

Таким образом, поиск решений в сфере обозначенных проблем представляет большой интерес всего сообщества. Иногда при создании подобных решений используется модель потоков данных (Dataflow, [8]). В различных вариантах методики, основанные на моделях потоков данных, применяются для создания процессорных архитектур, суперкомпьютеров в целом, для программной организации вычислительных потоков в рамках одного процесса и взаимодействия процессов в распределенной вычислительной среде.

В настоящей работе представлена методика программирования в распределенных вычислительных средах, имеющая целью устранить показанные проблемы.

Методика основана на анализе различных, в том числе и авторских, моделей потока данных, и возникла в результате продолжительной теоретической работы над архитектурой операционной системы для распределенных вычислений [9].

2. Требования к методике распределенных вычислений

Основные критерии, на которые ориентировались авторы при разработке описываемой методики, следующие:

1) методика должна предоставить универсальный механизм параллельного программирования, более простой в применении, чем существующие универсальные средства;
2) вычислительные программы, реализованные с помощью разрабатываемой методики, должны выполняться не менее эффективно, чем при использовании других средств;
3) методика должна ориентироваться на создание и эффективное исполнение программ на всех типах вычислительных систем, в первую очередь в распределенных вычислительных средах;

Под универсальными средствами параллельного программирования понимаются средства, способные обеспечить создание разнообразных параллельных программ, не привязанных к какому-либо методу обеспечения параллелизма. Это важное и серьезное требование к разрабатываемым средствам программирования, с одной стороны позволяющее решать максимально широкий спектр задач, но затрудняющее разработку этих средств с другой стороны.

Требование более простого режима использования, чем существующие универсальные средства, не нуждается в отдельном комментарии. Аналогично не нуждается в пояснении и требование эквивалентной эффективности исполнения вычислительных программ.

Обратимся к последнему критерию. Как отмечено во введении, существующие универсальные средства параллельного программирования, как правило, ориентированы на какой-то один конкретный тип вычислительной среды. Например, OpenMP разработан для программирования систем с общей памятью, MPI – для кластерных сред, Globus Toolkit – для грид-систем, OpenCL [10] – для управления нестандартными устройствами, в первую очередь GPU. (Система MPICH-G2 [11] не вполне соответствует «кластерному» MPI, т.к. требует особого внимания программиста к нюансам используемой распределенной среды)

Вместе с тем особый интерес представляет создание такого универсального средства, которое бы обеспечивало возможность разработки и эффективного исполнения программ во всех типах вычислительных сред, а также в их смешанных конфигурациях.

Наиболее сложной задачей из встречающихся в разработке параллельных программ является программирование распределенных вычислительных сред. В работе [12] показаны отличительные особенности данного типа сред и трудности их преодоления:

1. *Масштабность* – суммарное число процессоров, объем памяти и т.д. может на порядки превосходить показатели кластерных систем.
2. *Распределенность* – расстояние между вычислительными узлами может исчисляться тысячами километров с соответствующей шириной канала и латентностью.
3. *Неоднородность* – узлы могут обладать существенно различными характеристиками.
4. *Динамичность* – вычислительные узлы могут включаться и отключаться в произвольные моменты времени, более того, можно рассматривать узлы как ненадежные.
5. *Различная административная принадлежность*.

Разрабатываемая методика параллельного программирования должна учитывать эти особенности. Кроме того, необходимо учесть и аспекты, связанные с другими типами вычислительных сред.

При работе в кластерных средах необходимо учитывать и эффективно использовать наличие быстрых связей между узлами и возможно – быстрых параллельных файловых хранилищ. В системах с общей памятью – необходимо задействовать возможность доступа к данным без их копирования, складирования или передачи по сети. Разумеется, необходимо обеспечить и возможность совместного, смешанного варианта использования различных типов вычислительных систем: например, распределенная среда может состоять из набора кластеров, каждый из которых, в свою очередь, объединяет набор многоядерных машин с общей памятью.

В качестве дополнительного аспекта необходимо учесть и наличие таких устройств, как GPGPU, ПЛИС, Cell. Желательно, чтобы в рамках методики можно было бы поддержать программирование и данных устройств с помощью удобных средств.

Итак, создаваемая методика должна учитывать всю обозначенную выше совокупность аспектов работы различных вычислительных сред.

3. Методика RiDE

Методика базируется на понятиях ***хранилища***, ***задач*** и ***правил***.

Хранилище содержит в себе именованные данные, по отношению к которым доступны три операции – создание (запись), чтение и удаление. Хранимые данные являются самодостаточными - это не очереди, но некие цельные единицы информации с уникальными именами. Допускаются операции частичного чтения данных.

Задачей называется программа, которая во время исполнения считывает данные с определенными именами из *хранилища* и в результате своего исполнения формирует новые данные, которые записываются в хранилище.

Правилом называется такая конструкция, которая определяет условия и параметры запуска *задач*. Правило содержит в себе:

1. Список имен данных, которые необходимы для выполнения задачи.
2. Список соответствия глобальных имен данных (находящихся в хранилище) локальным именам (с которыми и будет работать задача).
3. Список задач (программ), которые необходимо запустить.
4. Действия, совершаемые в случае успешного выполнения задач (3).

Правило считается готовым к исполнению, когда в хранилище присутствуют все данные с именами из списка (1). После успешного исполнения правило удаляется из списка выполняемых правил.

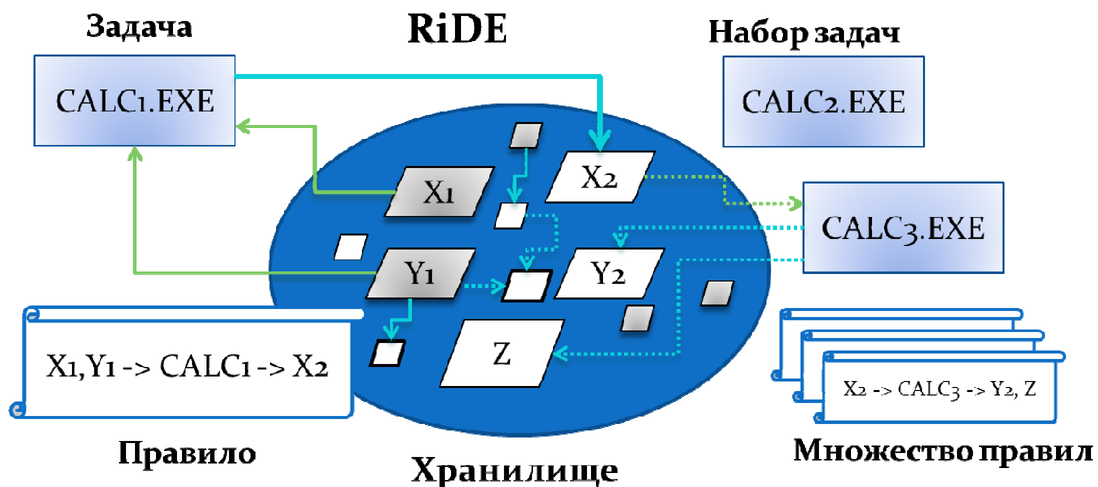

Рис. 1. Участники процесса вычислений в RiDE

На рис. 1 представлена общая схема участников описываемой методики. В центре находится хранилище. Белым выделены данные, которых в хранилище пока нет, серым - которые уже есть. Показан пример правила, которое гласит: при наличии данных X1 и Y1 необходимо запустить программу Calc1.exe, подать ей на вход эти данные, а результат работы записать в данные с именем X2. Показано и другое правило, которое требует выполнить другую программу при наличии элемента данных с именем X2. Очевидно, что это правило сработает только после того, как будет завершено первое правило. Это отмечено с помощью пунктира, который говорит, что обозначенный запуск и как результат порождение новых данных пока невозможно, но свершится в будущем. Вполне вероятно, что на рисунке есть и правила, которые могут выполняться независимо от представленных двух. Степень независимости и определяет меру параллелизма, с которой может быть произведено вычисление.

Процесс программирования и проведения вычислений в рамках RiDE происходит следующим образом. Прежде всего, разрабатываются программные коды задач, из которых состоит вычислительный эксперимент. Каждая такая задача на этапе инициализации должна считать данные из хранилища, а затем по ходу выполнения сформировать и записать новые данные в хранилище. Отметим, что в рамках одного вычисления могут использоваться любые комбина-

ции языков, а также целевых аппаратных сред для создания задач. Например, часть задач можно реализовать на графических ускорителях, а часть – на обычных процессорах.

После создания вычислительных программ (задач) программистом формируется файл инициализации, в котором описываются начальные правила системы. В дальнейшем эти правила могут дополняться – при выполнении задач или финализации правил. Кроме правил, в файле инициализации указываются начальные данные, которые помещаются в хранилище.

После подачи команды на запуск вычислительная среда ищет правила, готовые к исполнению, и запускает указанные в них задачи на подходящих свободных вычислительных ресурсах. В результате часть правил исполняется, формируя новые данные и освобождая ресурсы для других правил. Среда продолжает поиск и выполнение правил вплоть до исчерпания всех правил, приостановки работы с внешней стороны или выявления ошибки.

4. Преимущества, обеспечиваемые методикой RiDE

Программная реализация предлагаемой методики способна обеспечить следующие преимущества.

- Разделение уровней вычисления и взаимодействия, что обеспечивает более ясный процесс создания, отладки и сопровождения вычислительных программ. Вычислительные коды группируются в задачах; коммуникационные – в описаниях правил.
- Поддержка всех типов вычислительных сред – с общей памятью, кластерных и распределенных. Работа во всех средах может быть реализована эффективно, без необходимости переработки вычислительных программ, включая системы с общей памятью, где чтение и запись данных могут быть реализованы без накладных операций копирования.
- Возможность включения и отключения вычислительных ресурсов «на лету», что позволяет максимально эффективно задействовать вычислительные ресурсы и гибко планировать их распределение. Это свойство обеспечивается тем, что все обмены данными происходят через интерфейсы хранилища.
- Использование любых языков программирования для создания вычислительных программ. В программах должны присутствовать только два типа RiDE-функций – чтение и запись данных в хранилище.
- Отсутствие ограничений на внутреннюю сложность задач, вызываемых при срабатывании правил – задача, например, сама может быть параллельной MPI-программой или даже закрытой коммерческой программой, написанной вне рамок предлагаемой методики. В последнем случае взаимодействие программы с RiDE-окружением реализуется через файлы.
- Возможность участия в рамках одного вычисления программ различных платформ (различные ОС, языки программирования, процессоры и ускорители). Назначать правила на исполнение можно сообразно требованиям кодов задач к аппаратным характеристикам вычислительных узлов. Более того, в правиле можно указать различные версии вычислительных программ, написанные для разных целевых платформ; выбор конкретной версии может осуществляться исходя из имеющихся свободных вычислительных ресурсов.
- Возможность реализации поддержки программирования ускорителей. Программные средства уровня OpenCL [10] очень сложны в использовании; в рамках методики можно реализовать механизмы, упрощающие загрузку-выгрузку данных из вычислительных устройств;
- Встроенная поддержка контрольных точек. Весь обмен данными, и таким образом, текущее состояние счета, размещается в хранилище. Снимок состояния хранилища и формирует контрольную точку. При этом состоянием оперативной памяти считающихся в текущий момент задач можно пренебречь, осуществив при необходимости их пересчет.
- Возможность полностью остановить счет и в будущем продолжить его. Продолжение счета может быть осуществлено на других вычислительных ресурсах.
- Автоматическая оптимизация вычислений путем оптимального размещения правил на вычислительных узлах. Распределение правил по узлам может осуществляться на основе анализа а) статистики по предыдущим запускам, б) текущего размещения данных, в) зависимостей данных, описанных в правилах (заметим, что это описание – явное).

5. Вопросы реализации методики

Рассмотрим возможный вариант архитектуры системы, реализующей методику RiDE. Принципиально она состоит из двух уровней: среды выполнения и системы хранения. Среда выполнения отвечает за поиск готовых правил и их вычисление, система хранения отвечает за хранение всех данных, создаваемых и читаемых задачами. Также система хранения содержит и внутреннюю информацию, необходимую для работы системы в целом.

Среда выполнения состоит из следующих частей:

- ***Runner*** *(координатор)* – считывает файл исходных правил, выявляет готовые правила и размещает их для выполнения на доступных *Runsite*-узлах.
- ***Runsite*** *(область запуска)* – представитель вычислительного узла или группы узлов. Ведет учет доступных ресурсов (процессорных ядер, аппаратных ускорителей и т.д.), их типов и характеристик (архитектура, ОС, объем памяти и т.д.). Получает задания на запуск правил от модуля *Runner*.
- ***Runvisor*** *(трамплин)* – модуль, отвечающий за вычисление одного правила. *Runvisor* запускается модулем *Runsite* в результате размещения команд на выполнение правил.

Представленные участники системы и связи между ними изображены на рис. 2.

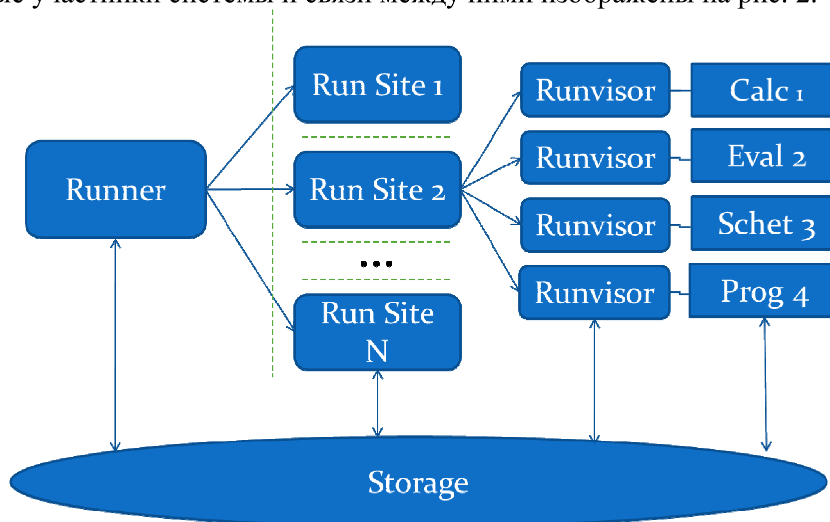

Рис. 2. Возможная архитектура системы

Модель исполнения в рамках данной архитектуры описывается следующим образом.

Модуль Runner считывает исходные правила. Также он считывает список адресов Runsite-узлов, доступных для использования. Runner определяет готовые правила, и производит поиск подходящего Runsite-узла среди множества доступных узлов. Искомый Runsite-узел должен обладать необходимым количеством свободных ресурсов и подходящими характеристиками (тип процессора, объем памяти и т.д.). Если подходящий свободный Runsite- узел обнаружен, то правило размещается на нем для исполнения. В случае, если на текущий момент таких узлов нет, то правило откладывается для последующих попыток исполнения.

При размещении правила на Runsite-узле этот узел а) производит синхронизацию вычислительных задач правила, получая при необходимости более новые версию (набор бинарных или текстовых файлов с вычислительными кодами) от модуля Runner и б) производит запуск процесса Runvisor и делегирует ему исполнение правила.

Runvisor производит процедуры подготовки запуска правила. Одним из важных этапов инициализации является настройка перенаправлений имен данных хранилища. Подразумевается, что в правилах указываются имена в глобальном пространстве имен. Однако вычислительные программы проще создавать, используя имена в локальном пространстве. Поэтому Runvisor должен позаботиться о том, чтобы передать запускаемым задачам информацию о соответствии глобальных и локальных имен.

Затем Runvisor производит запуск задач, указанных в правиле, и ожидает их завершения. В случае успешного завершения всех задач данного правила, Runvisor отмечает это правило как завершенное. При этом в ходе выполнения задач формируются новые данные, которые записываются в хранилище с применением глобальных имен. Также в ходе работы задач или на этапе финализации текущего правила могут формироваться новые правила. Таким образом, возможна генерация графа вычислений «на лету».

Вычисление считается законченным, когда Runner выявляет, что список правил, готовых к выполнению, исчерпался.

Предложенная архитектура среды выполнения опирается на распределенное хранилище данных. Рассмотрим вопросы создания такого хранилища.

6. Требования к распределенному хранилищу данных

Система хранения, необходимая для эффективной работы RiDE, должна удовлетворять определенным требованиям. Анализ существующих проектов хранилищ данных, включая различные распределенные базы данных, файловые системы, распределенные хэш-таблицы (DHT) показал, что в настоящий момент нет готовой системы, которая удовлетворяла бы всем выдвигаемым требованиям. В настоящее время авторский коллектив ведет проработку проекта распределенной системы хранения, удовлетворяющей следующим критериям:

1. Хранение как маленьких (от 1 байта), так и больших (несколько гигабайт) по объему элементов данных.
2. Хранение большого количества элементов данных (несколько миллиардов). Это связано с необходимостью поддержки вычислений, таких как сеточные расчеты; при этом необходимо хранить данные для нескольких последних итераций работы алгоритмов.
3. Работа в распределенном режиме, что включает поддержку неоднородности связей между узлами хранения.
4. Работа с неоднородными узлами и неодинаковой степени нагрузки. Большинство современных проектов распределенных хранилищ подразумевают равномерное распределение данных между узлами. Очевидно, что в случае RiDE такая политика является скорее вредной. Данные должны распределяться сообразно а) источникам этих данных, б) потребителям данных, в) возможностям машин, которые разнятся от узла к узлу.
5. Работа в многопользовательском режиме с различными пространствами имен. Требование связано с необходимостью поддержки одновременных расчетов, проводимых разными пользователями, и хранением результатов после проведения расчетов.
6. Возможность включения и исключения узлов хранения по ходу работы. С одной стороны, это требование касается условий ненадежности узлов. С другой стороны оно необходимо для реализации гибкого планирования и подключения узлов к вычислениям по ходу работы.
7. Использование отображения файлов в память для минимизации дисковых операций и разделяемой виртуальной памяти с механизмом копирования страниц при записи для дедубликации данных при одновременном доступе к одним и тем же данным из разных процессов.
8. Отслеживание появления новых данных с целью уведомления среды о готовности правил. В рамках одного вычисления в RiDE подразумевается наличие миллионов (возможно и миллиардов) экземпляров правил и соответственно, не меньшего числа данных. Соответственно для эффективного обнаружения готовых правил разумно использовать саму систему хранения, полностью или частично возложив на нее эту работу.
9. Эффективный поиск элементов данных по имени или части имени. Данное требование при реализации пункта (8) не является обязательным для RiDE, однако в общем случае такая функциональность была бы полезной.

7. Эксперимент по внедрению методики

Одним из хороших примеров применения методики RiDE является портирование с MPI rho-алгоритма Полларда для поиска дискретных алгоритмов. Основным преимуществом применения RiDE было сокращение исходного кода и упрощение логики параллельного выполнения.

Базовый алгоритм основан на поиске цикла (пересечения) в некоторой, специальным образом определенной, последовательности элементов, для чего используется алгоритм Флойда поиска циклов. Основным преимуществом алгоритма является минимальное требование к памяти - O(1), но он является очень сложным для параллельного вычисления и эффективность распараллеливания составляет sqrt(m), где m – количество вычислительных процессов.

Для оптимизации параллельной работы данного алгоритма была использована модификация rho-алгоритма Полларда со «специальными» точками, описанная в [14]. Данная модификация позволяет добиться практически линейного масштабирования, но требует большого объема памяти для эффективного выполнения (time-memory tradeoff).

Приведем краткое описание выполнения алгоритма для различных реализаций.

```
1. Выбрать начальные значения и некий критерий опреде-
   ления «специального» значения;
2. Вычислять значения последовательности пока не най-
   дется значение, подпадающее под критерий «специаль-
   ного» значения;
3. Проверить, существует ли данное значение в хранили-
   ще;
4. Если значения нет в хранилище - записать его в хра-
   нилище и перейти к пункту 2;
5. Вычислить результат.
```

Рис. 3. Линейный алгоритм

```
1. Выбрать начальные значения (различные для каждого
   процесса) и общий критерий определения «специально-
   го» значения;
2. Каждый процесс:
   2.1. Ищет новое «специальное» значение, независимо
        от других процессов;
   2.2. Проверяет новые, входящие точки, от других про-
        цессов (каждые несколько секунд) и ищет входя-
        щие значения в локальном хранилище. Если значе-
        ние найдено, то переходит к пункту 3;
   2.3. При нахождении нового «специального» значения
        процесс должен проверить, существует ли оно в
        локальном хранилище и перейти к пункту 3 если
        существует, либо отослать новое значение всем
        процессам в противном случае;
   2.4. Перейти к пункту 2.1;
3.  Вычислить результат.
```

Рис. 4. MPI реализация алгоритма

```
1. Выбрать начальные значения (различные для каждой
   задачи), получить параметры алгоритма из хранилища
   и выбрать общий критерий определения «специального»
   значения;
2. Каждая задача:
   2.1. Находит новое специальное значение;
   2.2. Ищет его в распределенном хранилище данных и
```

```
          переходит к пункту 3 если такое значение суще-
          ствует;
    2.3. Порождает новое правило повторения итерации и
          заканчивает работу;
 3. Вычислить результат.
```

Рис. 5. RiDE реализация

Приведенные выше алгоритмы показывают, что реализация с использованием методики RiDE позволила:

- Существенно облегчить логику каждой отдельной задачи;
- Исключить необходимость пересылки и проверки новых входящих значений между процессами;
- Сократить размер кода. На практике, более 60% кода было просто удалено и заменено несколькими клиентскими вызовами RiDE на чтение и запись данных;

Все изменения не повлияли на производительность исходной реализации алгоритма [14].

8. Заключение

В работе представлена методика параллельного программирования, которая может быть использована при программировании распределенных вычислительных сред, кластерных систем и систем с общей памятью.

Методика позволяет достаточно просто и эффективно реализовать проведение вычислительного эксперимента на гибридных архитектурах, позволяет осуществлять динамическое изменение количества вычислительных узлов по ходу вычислений, автоматизирует создание контрольных точек, приостановку и продолжение вычислений прозрачным для программиста образом, а также обеспечивает ряд других преимуществ.

Одной из целей, которые были поставлены при создании методики, являлось упрощение процесса создания суперкомпьютерных вычислительных программ, по отношению к средствам класса MPI и OpenMP. Первые эксперименты показывают, что методика достигла этой цели.

На основе предложенной методики авторами прорабатываются вопросы реализации среды параллельного программирования. Первые эксперименты показывают реализуемость предлагаемых идей и лаконичность программных конструкций для описания правил. Авторы выражают уверенность, что в результате развития этой среды удастся достичь главной цели – сделать процесс создания распределенных вычислительных программ более простым и эффективным.

Информация о ходе исследования размещается на сайте www.ridehq.net.

Литература

1. Dongarra J. J., Hempel R., Hey A. J. G., and Walker D. W. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report TM-12231 // Oak Ridge National Laboratory, February 1993.
2. OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface 1.0, 1997.
3. Foster I., Kesselman C. Globus: A Metacomputing Infrastructure Toolkit. // Intl Journal Supercomputer Applications, 1997, 11(2): P. 115-128.
4. Konovalov N.A., Krukov V.A., Mihailov S.N. and Pogrebtsov A.A. Fortran DVM - a Language for Portable Parallel Program Development // Proceedings of Software For Multiprocessors & Supercomputers: Theory, Practice, Experience, Institute for System Programming, RAS, Moscow, 1994, P. 124-133.

5. Абрамов С.М., Васенин В.А., Корнеев В.В., Московский А.А., Роганов В.А. Организация распределенной общей памяти в Т-системе с открытой архитектурой (рус.) // ИПС РАН, ЦНТК. — 2003. — С. 13.
6. Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters // OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
7. Michael Voss. Demystify Scalable Parallelism with Intel Threading Building Block's Generic Parallel Algorithms // DevX.com, Jupiter Media, October 2006.
8. Dennis J. Data Flow Supercomputers // Computer, 1980, Vol.13, No.11, P.48-56.
9. Бахтерев М.О. Описание параллельных вычислений при помощи замыканий // Тезисы 10-го Международного семинара "Супервычисления и Математическое моделирование", РФЯЦ-ВНИИЭФ, Саров, 2008, С. 31-32.
10. Jaejin Lee et al. An OpenCL framework for heterogeneous multicores with local memory // In PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques, 2010, P. 193-204.
11. Karonis N., Toonen B., and Foster I. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface // Journal of Parallel and Distributed Computing, 2003, P. 1-22.
12. Воеводин Вл.В. Решение больших задач в распределенных вычислительных средах. //Автоматика и Телемеханика. 2007, N5, С. 32-45.
13. Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications // 1996 September 23, P. 1-30.
14. Albrekht I. Some methods for DLP solving, and time estimation // Information Security Conference, Yekaterinburg, 2005, P. 20-25.