Программная архитектура для использования параллельных вычислений при численном балансировании потоков газа на расчетном графе газотранспортной системы

Васильев А.В., Сарданашвили С.А.

РГУ нефти и газа имени И.М.Губкина

Введение

Любая технологическая газотранспортная система может быть представлена «расчетной схемой»: графом, дугами которого являются моделируемые объекты: трубопроводы, краны, перемычки, компрессорные цеха, отводы к ГРС и так далее [1]. Вершинами графа являются соединения входов и выходов расчетных объектов.

В общем виде, «моделируемые объекты» это любые объекты, режим которых моделируется расчётной процедурой

$$\varphi\left(\overrightarrow{Passport}, \overrightarrow{GasIn}, \overrightarrow{GasOut}\right),$$

где:

$$\overrightarrow{Passport} -$$

набор паспортных параметров объекта, включая его режимно-технологические ограничения,

$$\overrightarrow{GasIn} = \{PIn, TIn, QIn\},$$

$$\overrightarrow{GasOut} = \{POut, TOut, QOut\} -$$

параметры газовых потоков на входе/выходе объекта (часть из которых в модели заданы, часть - расчётные). P – давление, T –температура, Q – расход.

Для решения многих задач планирования и управления режимами ГТС используется алгоритм «балансирования потоков» газа в узлах ГТС, представленный в работе [1]. В его основе лежит задача «расчёта режима моделируемого объекта»:

$$\{q_{\text{вх}}, q_{\text{вых}}, T_{\text{вых}}\} = f(\overrightarrow{Passport}, P_{\text{вх}}, P_{\text{вых}}, T_{\text{вх}}),$$

которую приходится многократно решать для всех объектов расчётной схемы ГТС в процессе выполнения алгоритма.

Поскольку данный алгоритм является итерационным, количество «моделируемых объектов» (дуг графа) может достигать нескольких тысяч, то при последовательном выполнении процедур расчёта режимов объектов ГТС суммарное время вычислений может оказаться неприемлемым. Особенно это утверждение справедливо при моделировании нестационарных режимов. Для того чтобы полностью использовать потенциал современного вычислительного оборудования, и тем самым производить решение задач с максимальной скоростью, необходимо применение параллельных вычислений [2,3,4,5].

Современные вычислительные устройства, производимые такими компаниями как Intel, AMD, ATI, nVidida, содержат параллелизм в своей аппаратной архитектуре. Активно разрабатываются, внедряются и популяризируются инструменты, позволяющие разработчикам программного обеспечения задействовать в своих продуктах предоставляемые аппаратные возможности. Для организации параллельных вычислений в настоящее время активно используются следующие технологии [6,7]:

- Использование многоядерных процессоров с помощью многопоточности, OpenMP, библиотек параллельных примитивов от Microsoft, Intel.
- Вычисления общего назначения на графических адаптерах General Purpose Graphics Processing Unit (GPGPU) с помощью технологий CUDA, OpenCL.
- Осуществление распределённых вычислений на кластерных системах с использованием передачи сообщений по интерфейсу MPI (Message Passing Interface).
- Организация гетерогенных параллельных вычислений – комбинация разнородных вычислительных технологий, например, совместное использование вычислений

на многоядерных центральных процессорах и массивно-параллельных сопроцессорах.

Следует отметить, что хотя перечисленные технологии являются популярными, доступными, распространёнными, их использование в программных продуктах, которые не были изначально для этого приспособлены, может быть сопряжено со значительными трудностями. Важнейшее значение приобретает архитектура приложений, позволяющая свободно использовать технологии параллельных вычислений для достижения высокой производительности.

Требования к разрабатываемой архитектуре и принципы, позволяющие их выполнить

Программная архитектура расчетного режимно-технологического комплекса должна отвечать следующим требованиям:

1. *Гибкость* - возможность легко изменять, добавлять новые расчетные модели объектов, выбирая для той или иной модели наиболее подходящую технологию параллельных вычислений. Важность требования гибкости подтверждается тем, что, во многом, именно её отсутствие не даёт сейчас многим ПВК использовать современные технологии.
2. *Высокая производительность* - возможность использования параллельных вычислений.
3. *Масштабируемость* - возможность наращивать производительность по мере необходимости, добавляя вычислительные мощности, не перерабатывая расчетных алгоритмов.

В объектно-ориентированном программировании выработан набор принципов, позволяющий удовлетворять подобным требованиям, он называется SOLID [8,9,10]. Это аббревиатура, означающая следующие принципы:

- Single Responsibility Principle (SRP) – принцип единственной ответственности: у класса должна быть одна чётко определённая ответственность.

- Open-Closed Principle – архитектура приложения должна быть открыта для расширения, но закрыта для изменения.
- Liskov Substitution Principle – принцип подстановки Лисков. Каждый унаследованный объект должен быть свободно подставляем вместо родительского.
- Interface Segregation Principle – принцип разделения интерфейсов. Предпочтительнее иметь набор маленьких интерфейсов, чем один интерфейс, содержащий обширное множество методов.
- Dependency Inversion Principle – принцип инверсии зависимостей.

Для построения высокопроизводительной, гибкой, масштабируемой программной архитектуры решения поставленной задачи целесообразно выделить следующие программные модули:

1. Модуль, предоставляющий интерфейс для «манипулирования» графом.
2. Модуль вычислительных процедур, выполняемых с помощью интерфейсных методов модуля «манипулирования» графом.
3. Модуль организации параллельного расчёта объектов.

Для обеспечения требуемой гибкости необходимо соблюсти принцип инверсии зависимостей: «модули высокого уровня не должны зависеть от модулей низкого уровня, и те и другие должны зависеть от абстракций», и связать перечисленные модули через абстракции [11].

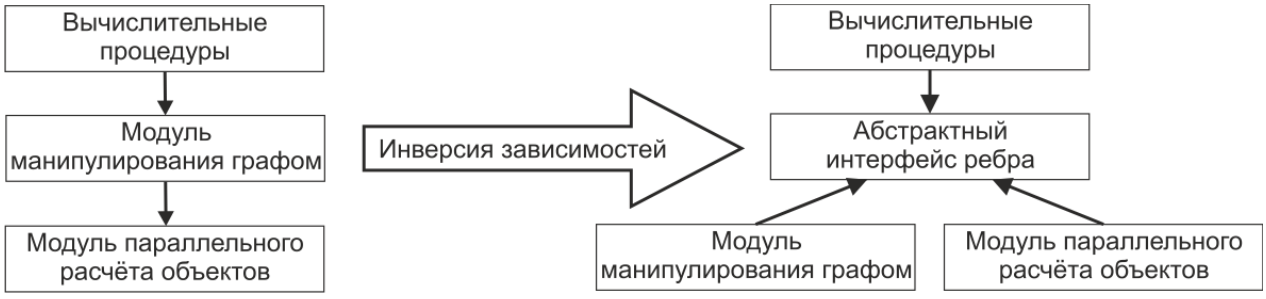

Рисунок 1. Взаимосвязи модулей до и после применения принципа инверсии зависимостей.

Связующей абстракцией является абстрактный интерфейс дуги графа – *Edge*, содержащий в соответствии с определением «моделируемого объекта», содержит следующие методы:

1. void SetPIn(double p_in) - задать давление газа на входе «моделируемого объекта».
2. void SetPOut(double p_out) - задать давление газа на выходе «моделируемого объекта».
3. double GetQ() – рассчитать значение расхода по дуге графа на основе модели объекта-дуги.

Далее более подробно рассмотрен модуль организации параллельного расчёта моделируемых объектов.

Менеджеры параллельного расчёта

Для каждого типа «моделируемых объектов» создаётся класс, отвечающий за параллельный расчёт объектов этого типа с использованием наиболее подходящей технологии параллельных вычислений, называемый «Менеджером параллельного расчёта». Этот класс инкапсулирует следующие функции:

1. Создание объекта по его паспортным данным, и предоставление доступа к методам созданного объекта через абстрактный интерфейс ребра графа.
2. Организация эффективного хранения данных объектов в памяти, их удаление при завершении работы.
3. Организация параллельного расчёта объектов.

Абстрактный интерфейс «менеджера параллельного расчёта» предоставляет следующие методы:

1. Edge* Create(const Passport& passport) – создание объекта по паспорту и возврат указателя на абстрактный интерфейс ребра.

2. void CountAll( ) – произведение расчётов всех управляемых объектов с использованием параллельных вычислений. В результате выполнения этой функции метод Edge->GetQ() возвращает актуальное значение расхода по объекту.

Исходя из эмпирических соображений, можно сказать, что в зависимости от типа моделируемого объекта, вычислительной сложности его математической модели, среднего количества объектов данного типа, лучше может подходить та или иная технология организации параллельных вычислений.

Тип объекта	**Количество объектов**	**Сложность моделей**	**Применимая технология параллельного расчёта**
Трубопровод, кран	Большое	Низкая	Массивно-параллельные расчёты, GPGPU
Газоперекачивающий агрегат, компрессорный цех	Среднее	Средняя	Расчёты с использованием многоядерных процессоров, библиотек параллельных примитивов
Трубопроводная система	Низкое	Высокая	Использование вычислительного кластера, MPI

Таблица 1. Целесообразность применения различных технологий параллельных вычислений для «моделируемых объектов» различных типов.

Для различных технологий параллельных вычислений лучше или хуже может подходить та или иная стратегия размещения данных в памяти. Для организации вычислений на многоядерном процессоре подходит стратегия «Array Of Structures» (AoS) – массив структур. Структуры с данными, необходимыми для расчёта объектов, располагаются в памяти последовательно, что обеспечивает эффективную работу процессорного кэша [12].

Для реализации массивно-параллельных вычислений на GPGPU больше подходит противоположная стратегия хранения данных – «Structure Of Arrays» (SoA) – структура массивов [12,13].

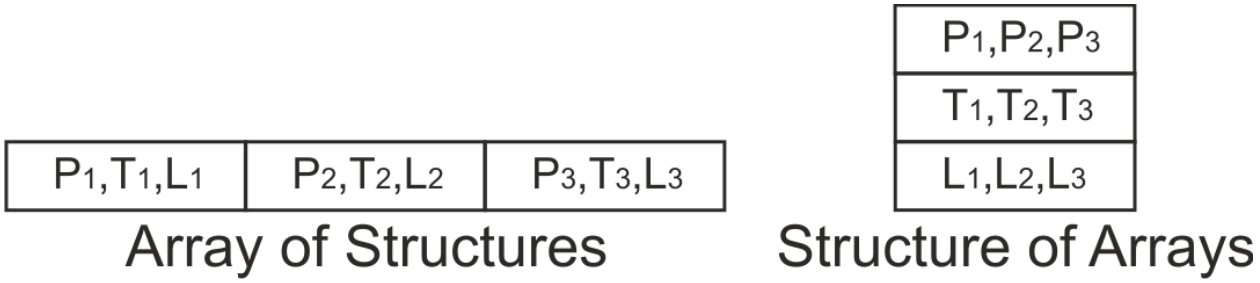

Рисунок 2. Различные стратегии размещения данных в памяти.

Размещение данных по стратегии SoA позволяет увеличить скорость чтения и записи в память GPU за счёт объединения запросов чтения и записи в группы. Для названия этого явления используется термин «coalescing». В зависимости от того, как расположены данные, скорость работы с памятью видеокарты может различаться в десятки раз [13,14,1].

Общее описание разработанной архитектуры

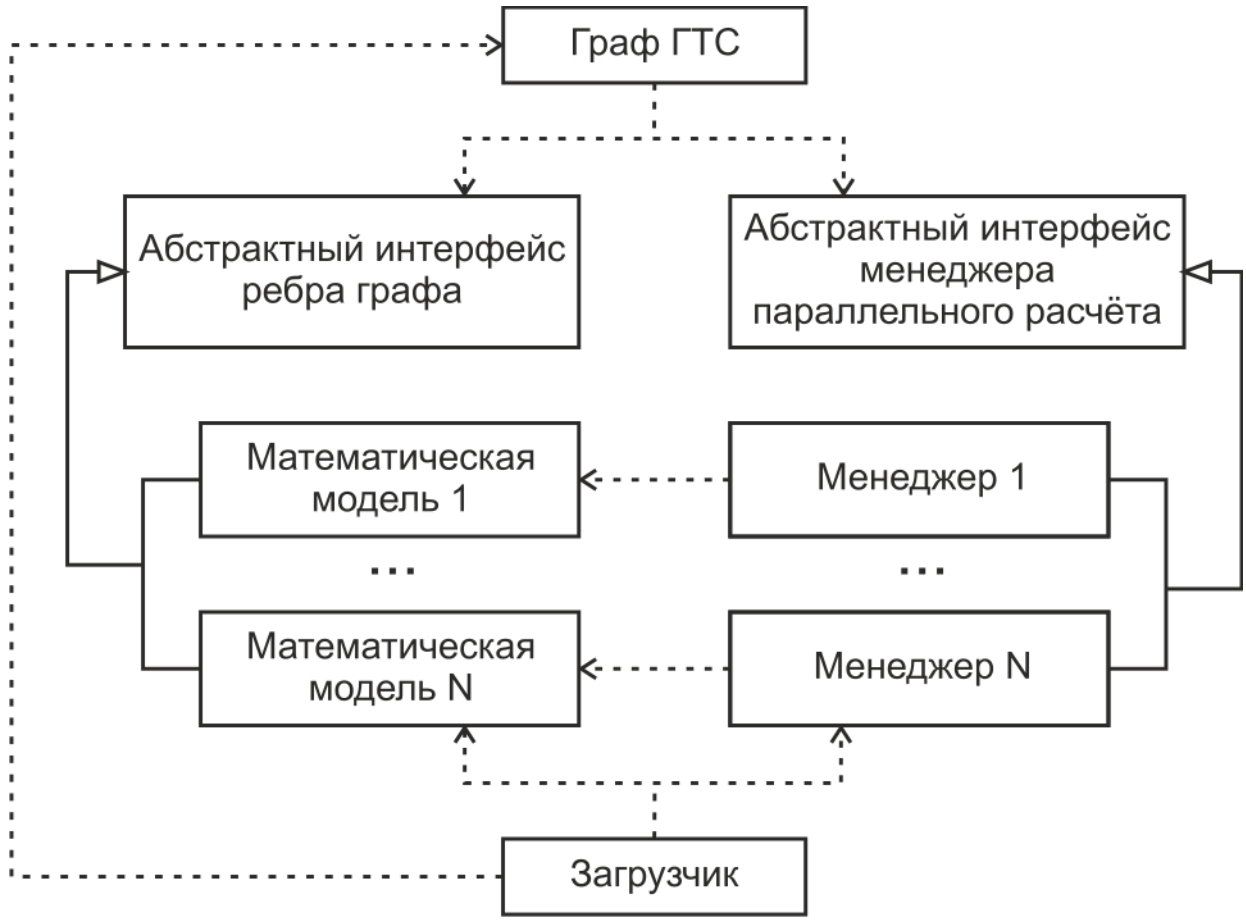

Рисунок 3. Общая схема разработанной архитектуры.

Класс Загрузчик считывает информацию о «моделируемой схеме» из файла или базы данных и строит граф соответствующей топологии, используя интерфейсные методы модуля «манипулирования» графом. При этом все «моделируемые объекты» создаются «менеджерами параллельного расчёта» соответствующих типов, указатели на интерфейсные объекты абстрактного типа *Edge* заносятся в соответствующие дуги графа.

Вычислительные процедуры работают с методами *Edge*, не имея никакой информации о том, какие объекты отвечают тем или иным дугам, и каким образом эти объекты будут рассчитываться параллельно.

Загрузчик ассоциирует с созданным графом информацию о «менеджерах параллельного расчёта», задействованных в управлении содержащихся в нём «моделируемых объектов», достаточную для запуска параллельного расчёта объектов в нужный момент.

Примеры «менеджеров параллельного расчёта»

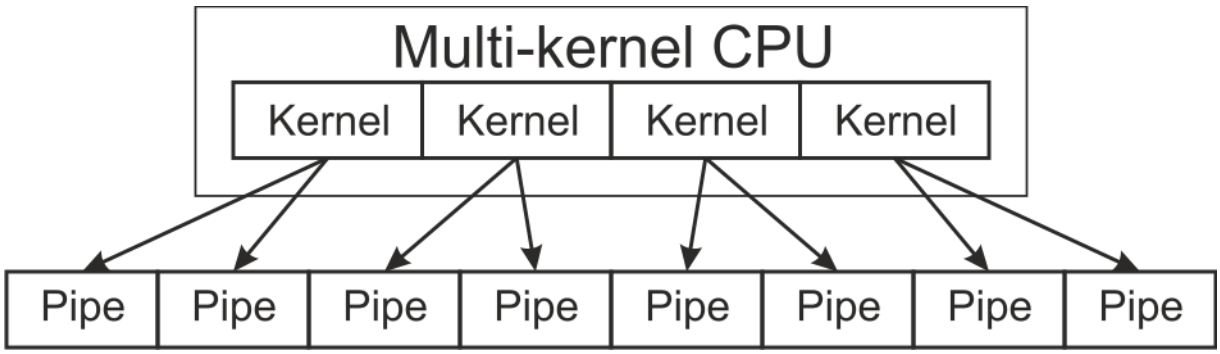

Рисунок 4. Схема «менеджера», использующего многоядерный параллелизм.

Для реализации «менеджера параллельного расчёта», иллюстрирующего применение технологии многоядерного параллелизма, для объектов типа «Трубопровод» был разработан класс, содержащий свойства и методы, необходимые для моделирования объектов этого типа. Функции «менеджера» в данном случае сводятся к созданию очередного передаваемого ему объекта по паспорту, добавлению созданного объекта в вектор, возврату указателя на этот объект и реализации метода параллельного расчёта множества управляемых объектов.

Хранение объектов в виде вектора соответствует стратегии Array of Structures и хорошо подходит для многоядерного параллелизма, обеспечивая эффективную работу процессорного кэша.

Реализация метода параллельного расчёта с использованием библиотеки Microsoft PPL имеет следующий простой вид:

```
#include <ppl.h>
Concurrency::parallel_for_each(edges_.begin(),edges_.end(),
[](Pipe &edge)
{ edge.Count(); } );
```

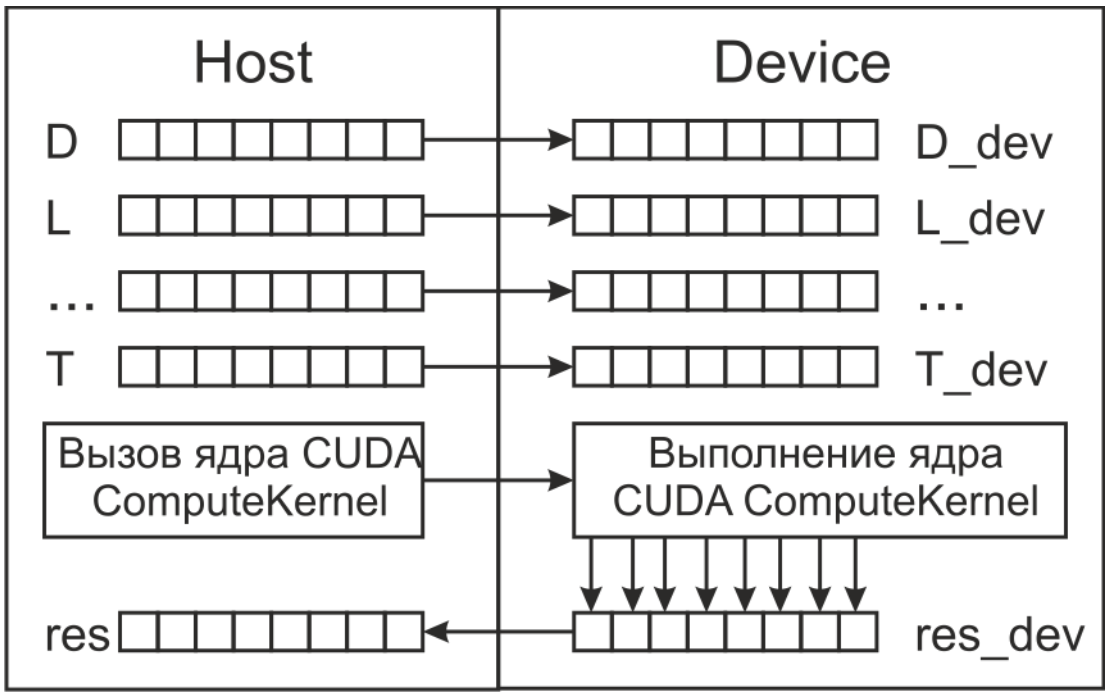

Рисунок 5. Схема «Менеджера», использующего массовый параллелизм на CUDA.

Для иллюстрации использования технологии массивно-параллельных расчётов на GPGPU был разработан «менеджер параллельного расчёта», осуществляющий расчёты на вычислительных устройствах архитектуры CUDA. Для организации эффективной работы с памятью CUDA GPU необходимо реализовывать стратегию размещения данных Structure Of Arrays, чтобы достичь эффекта «coalescing».

В предложенной реализации, в соответствии со стратегий SoA, менеджер собирает данные передаваемых ему под управление объектов в структуру векторов, для каждого переданного объекта возвращает указатель на абстрактный интерфейс ребра. Реализация интерфейсных функций ложиться на «менеджер», так как класса, отвечающего управляемому «моделируемому объекту» не существует.

Для обеспечения параллельного расчёта, менеджер копирует собранные вектора параметров в память устройства, конфигурирует и вызывает ядро CUDA, выполняющее «расчёт моделируемых объектов», копирует результат расчёта в память. Менеджер автоматически определяет количество доступных в системе CUDA GPU, разделяет данные на части, пропорциональные их производительности, и равномерно распределяет работу.

Организация вычислительного эксперимента по сравнению производительности различных менеджеров и анализ его результатов

Для оценки эффективности предложенной архитектуры был поставлен вычислительный эксперимент. Был сформирован граф расчетной схемы, содержащий 15000 рёбер, и произведен параллельный расчет режимов всех его объектов с использованием различных «менеджеров параллельного расчёта». Для простоты, в качестве расчетной модели дуги была выбрана модель трубопровода. Для того чтобы сгладить статистическую погрешность, вычисления производились многократно и усреднялись.

Вычислительные эксперименты производились на следующей аппаратной конфигурации – ЦП Intel Core i7 (8 ядер), две видеокарты nVidia GeForce GTX460.

В эксперименте участвовали 4 разработанных «менеджера параллельного расчёта» для расчёта модели трубопровода последовательным счётом [1]:

1. Менеджер, использующий 1 ядро центрального процессора.
2. Менеджер для многоядерного параллелизма с использованием библиотеки Microsoft Performance Primitives Library, использующий все 8 ядер центрального процессора.
3. Менеджер для CUDA GPU с выключенной поддержкой масштабирования – использующий только один GeForce GTX 460.
4. Менеджер для CUDA GPU с включённым автоматическим масштабированием на несколько GPU, использующий оба установленных GeForce GTX 460.

Менеджер	**Среднее время работы**	**Ускорение по отношению к использованию одного ядра**
С использованием одного ядра	7.02 (сек)	1 (x)
С использованием 8 ядер с PPL	1.17 (сек)	6 (x)
С использованием одного CUDA GPU	0.23 (сек)	30 (x)

С использованием 2 CUDA GPU	0.11 (сек)	60 (x)

Таблица 2. Результаты численного эксперимента.

Анализ результатов проведённого эксперимента показывает 30-кратное превосходство в скорости одного CUDA GPU над одним ядром центрального процессора, однако, в общем случае такое утверждение не всегда является верным по следующим причинам:

1. Возможно создание более производительного кода для процессоров Intel с использованием SSE, различными оптимизациями. То же относится к оптимизации кода CUDA: в зависимости от усилий, вложенных в оптимизацию, результат может существенно отличаться.
2. Соотношение мощности центрального процессора и графического адаптера может различаться для разных систем.
3. Для разных математических моделей может лучше или хуже подходить использование массивно-параллельной архитектуры CUDA.

Благодаря способности «менеджера» автоматически распределять нагрузку по доступным вычислительным устройствам, подключение второго CUDA-устройства позволило вдвое увеличить производительность. Такая масштабируемость позволяет увеличивать производительность простым добавлением аппаратных мощностей. Тем самым достигается возможность подбирать необходимую аппаратную конфигурацию вычислительной системы, исходя из требований к производительности.

Результаты эксперимента показывают, что разработанная архитектура отвечает требованиям высокой производительности и масштабируемости.

Возможность лёгкого использования для одного и того же «моделируемого объекта» четырёх разных «менеджеров параллельного расчёта» подтверждает гибкость архитектуры, достигнутой за счёт использования принципа инверсии зависимостей – модуль предоставления интерфейса манипулирования графом не зависит непосредственно от модулей организации параллельных расчётов объектов, они связаны через абстрактные

интерфейсы, и поэтому достаточно независимы. Это даёт возможность легко добавлять различные модели объектов в систему, эффективно организуя их параллельный расчёт наиболее подходящим способом.

Заключение

Гибкость, высокая производительность, масштабируемость предложенной архитектуры обеспечивают возможность разработки различных менеджеров параллельных расчётов, наиболее подходящих для некоторого типа «моделируемых объектов», позволяя переходить к использованию наиболее современных технологий параллельных вычислений по мере их развития.

Эффективным может быть комбинирование уже разработанных менеджеров. Если в вычислительной системе реализовано несколько различных менеджеров, способных работать одновременно, задействуя различные вычислительные устройства, например, использующих центральный процессор и видеокарту, то вычислительная работа может быть разделена на части, пропорциональные их производительности. Это позволит организовать гетерогенные параллельные вычисления и ещё больше увеличить производительность. В «менеджере», организующем расчёт сложных «моделируемых объектов» на вычислительном кластере [16] можно использовать менеджеры, эффективно реализующие расчёты простых объектов, входящих в состав сложных объектов.

Подходы, использованные при разработке предложенной архитектуры, показали свою эффективность в поставленном вычислительном эксперименте, они могут быть использованы при разработке общей архитектуры программно-вычислительного комплекса решения задач моделирования, оптимизации и прогнозирования режимов газотранспортных систем.

Авторами планируется использовать описанную архитектуру при разработке расчётного ядра нового поколения для комплекса моделирования режимов транспорта газа ПВК «Веста» [1].

Литература

1. *Сарданашвили С. А.* Расчётные методы и алгоритмы (трубопроводный транспорт газа). М.:ФГУП Изд-во «Нефть и газ» РГУ нефти и газа им. И.М. Губкина, 2005.
2. *Moore, Gordon E.* Cramming more components onto integrated circuits // Electronics Magazine. – 1965. – P. 4-8.
3. 1965 – "Moore's Law" Predicts the Future of Integrated Circuits // Computer History Museum. - 2007.
4. *Herb Sutter*. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software // Dr. Dobb's Journal. – 2005. - №.30(3).
5. *Shekhar Borkar, Andrew A. Chien*. The Future of Microprocessors // Communications of ACM. – 2011. - №.54 (5).
6. *T. Rauber, G. Runger*. Parallel programming (2nd edn.). Springer-Verlag Berlin Heidelberg 2010.
7. Parallel computing. [Электронный ресурс] // Wikipedia. - URL:http://en.wikipedia.org/wiki/Parallel_computing (дата обращения: 18.09.2011).
8. Robert Martin. Design principles and design patterns. [Электронный ресурс] // Object Mentor. - URL:http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf (дата обращения: 18.09.2011).
9. Robert Martin. The principles of OOD. [Электронный ресурс] // But Uncle Bob. - URL:http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod (дата обращения: 18.09.2011).
10. Robert Martin. Getting a SOLID start. [Электронный ресурс] // Object Mentor. - URL:http://blog.objectmentor.com/articles/2009/02/12/getting-a-solid-start (дата обращения: 18.09.2011).
11. *Robert Martin*. The Dependency Inversion Principle // C++ Report. – 1996. - May.

12. Stream processing. [Электронный ресурс] // Wikipedia. URL:http://en.wikipedia.org/wiki/Stream_processing (дата обращения: 18.09.2011).
13. *Боресков А. В., Харламов А. А.* Основы работы с технологией CUDA. – М.: ДМК Пресс, 2010.
14. *Jason Sanders, Edward Kandrot*. CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley, 2010.
15. *David Kirk, Wen-mei Hwu*. Programming massively parallel processors. Elsevier Inc, 2010.
16. Computer cluster. [Электронный ресурс] // Wikipedia. - URL: http://en.wikipedia.org/wiki/Cluster_(computing) (дата обращения: 18.09.2011).