УДК 004.272.2

С.Н. Верзунов
Институт автоматики и информационных технологий Национальной академии наук Кыргызской Республики, г.Бишкек
verzunov@hotmail.com

ОБЗОР МЕТОДОВ РАЗРАБОТКИ ПРИЛОЖЕНИЙ ДЛЯ ГЕТЕРОГЕННЫХ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

В работе даны краткие теоретические сведения и приведен обзор актуальных методов и технологий создания программ для гетерогенных параллельных вычислительных систем.

Ключевые слова: параллельное программирование, гетерогенная вычислительная система, графический процессор, уровни параллельности, гранулярность, синхронизация потоков.

Введение. В настоящее время вычислительные системы становятся все более сложными и гетерогенными в основном за счет использования не только традиционных центральных процессоров, но и многоядерных микропроцессоров, цифровых процессоров обработки сигналов [1], программируемых логических интегральных схем и, наконец, графических процессоров (Graphical Processor Unit, GPU). За счет такого разнообразия параллельных вычислительных устройств значительно усложняется процесс разработки эффективного программного обеспечения. Разрабатываемые приложения, как правило, создают неоднородную нагрузку на вычислительные системы, начиная от простых задач обработки данных, таких как, например, поиск, сортировка и синтаксический анализ, и заканчивая процедурами, требующими длительных математических вычислений – обработка изображений, моделирование и интеллектуальный анализ данных. В последнем случае эффективная работа приложений в большой степени зависит от вычислительной эффективности используемого оборудования. В качестве примеров таких задач можно привести:

- задачи молекулярной динамики,
- модели погоды и океанических течений,
- моделирование дрейфа тектонических платформ,
- моделирование роста клеток,
- моделирование распространения звука и световых волн [2].

Как правило, многие из этих задач могут решаться параллельно. Учитывая тепловые пределы современной технологии изготовления процессоров на основе комплементарных металлооксидных полупроводников, дальнейшее увеличение их производительности путем увеличения тактовой частоты представляется невозможным и оказывается более целесообразным разместить несколько процессоров, иногда специализированных, на одном чипе. В связи с этим за последние 10 лет значительно увеличились разнообразие и возможности параллельных вычислительных устройств. Возросла роль

графических процессоров, которые теперь обеспечивают высокую скорость обработки данных при очень низких затратах. Современные GPU обладают значительно большей вычислительной мощностью, чем любые другие программируемые устройства общего назначения. Разница в пиковой производительности по сравнению с мощными центральными процессорами персональных компьютеров достигает 20–50 раз, при том, что потребляемая мощность больше всего в 2 раза, а цена сопоставима. При этом их можно использовать для решения широкого круга задач, включая задачи цифровой обработки сигналов [3]. Уже имеются реализации самых различных алгоритмов для гетерогенных многопроцессорных вычислительных систем, содержащих GPU, включая БПФ, сортировку, алгоритмы линейной алгебры, всевозможные алгоритмы обработки изображений и видео, алгоритмы компьютерного зрения и многие другие [4]. На рис. 1 показано, как часто встречающиеся задачи, такие как сортировка и скалярное произведение векторов, могут быть решены с использованием множества процессоров [2].

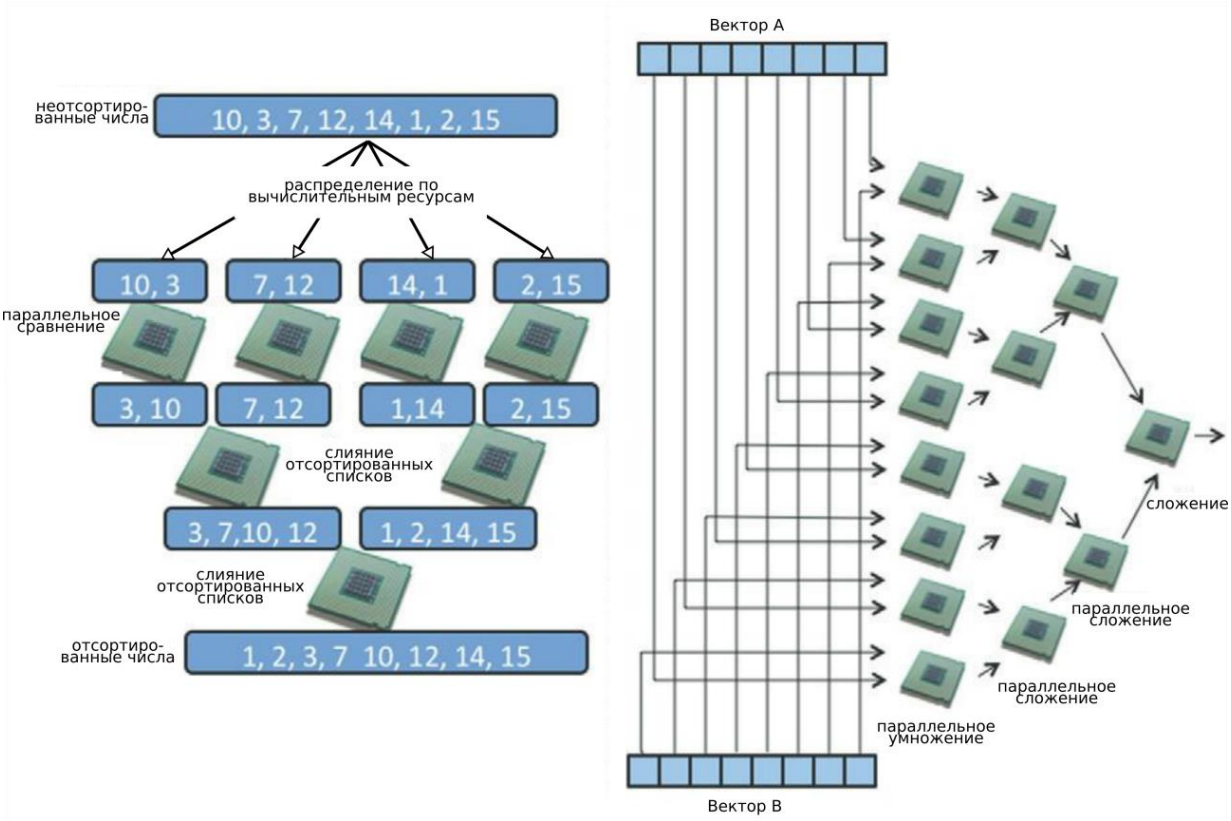

Рисунок 1. Параллельная сортировка и скалярное произведение векторов

В настоящее время существует несколько программных платформ для реализации параллельных вычислений, например CUDA для видеокарт фирмы NVIDIA [5], ATI Stream Technology для видеокарт от AMD [6]. Особое место занимает OpenCL (Open Computing Language, открытый язык вычислений) – программная платформа, сопровождаемая некоммерческим технологическим консорциумом Khronos Group [7]. Эта платформа применяется для разработки приложений, выполняющихся на широком спектре устройств различных производителей, поддерживает множество уровней параллелизма и может эффективно работать на гомогенных или гетерогенных, однопроцессорных и многопроцессорных вычислительных системах, содержащих, кроме тра-

диционных, графические процессоры и другие типы устройств. В настоящее время она поддерживает все основные архитектуры процессоров, включая x86, ARM и PowerPC, также она была адаптирована к драйверам графических плат AMD и NVIDIA. Поддержка среди производителей процессоров, видеокарт и ПЛИС быстро растет, что делает OpenCL перспективной платформой для разработчиков программного обеспечения, при этом можно быть уверенным в том, что она будет использоваться в течение многих лет, а сфера применения будет постоянно расширяться [3].

Таким образом, современные вычислительные системы характеризуются наличием разнообразных по своим возможностям и аппаратной реализации параллельных вычислительных ресурсов, что значительно затрудняет разработку для них эффективного программного обеспечения. В настоящей работе дан обзор актуальных методов и технологий создания программ для такого рода гетерогенных параллельных вычислительных систем.

Методы создания программ для параллельных вычислительных систем. С увеличением потребности в высокопроизводительных вычислениях и появлением соответствующего аппаратного обеспечения получили развитие различные методы ускорения вычислений за счет использования нескольких процессоров. Первый метод, использующий параллелизм на уровне задач, включает подходы, итеративно разбивающие проблему на подзадачи до тех пор, пока они не будут хорошо соответствовать имеющимся вычислительным ресурсам. Другой часто используемый метод, основанный на параллелизме уровня данных, подразумевает, что исходные данные разбиваются на подмножества, каждое из которых обрабатывается на отдельном процессоре, затем обработанные данные объединяются и формируют результат. При разбиении, как в первом, так и во втором случае учитывается размер подмножеств, так как необходимо, чтобы он соответствовал имеющимся аппаратным вычислительным ресурсам.

Параллельные вычисления [8] являются «одной из форм расчетов, в которых многие вычисления проводятся одновременно, исходя из принципа, что большие проблемы часто можно разделить на более мелкие, которые затем решаются одновременно», т. е. параллельно. «Степень параллелизма, которая может быть достигнута, зависит не только от характера рассматриваемой проблемы, но и умения разработчика программного обеспечения найти правильный подход к рассматриваемой задаче.» Джин Амдал [9] сформулировал ограничение на рост производительности при распараллеливании вычислений: «В случае, когда задача разделяется на несколько частей, суммарное время ее выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента». Согласно закону Амдала, ускорение выполнения программы за счет распараллеливания части ее инструкций на наборе вычислительных элементов ограничено временем, необходимым для выполнения ее последовательных инструкций.

Пусть необходимо решить некоторую вычислительную задачу. Доля α от общего объема необходимых вычислений может быть получена только с помощью последовательных расчетов и тогда доля 1 – α может быть распараллелена так, чтобы время ее вычисления было обратно пропорционально числу задействованных вычислительных элементов p. В этом случае ускорение, которое может быть получено на вычислитель-

ной системе из p элементов, по сравнению с решением с помощью одного элемента, не будет превышать величины:

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}.$$

В таб. 1 показано, во сколько раз быстрее выполнится программа с долей последовательных вычислений α при использовании p процессоров.

Таблица 1. Иллюстрация закона Амдала

α \ p	10	100	1000
0	10	100	1000
10%	5.263	9.174	9.910
25%	3.077	3.883	3.988
40%	2.174	2.463	2.496

Из таблицы видно, что только при полном отсутствии последовательных вычислений (т.е. при $\alpha = 0$) можно получить линейный прирост производительности с ростом количества вычислителей в системе. Закон Амдала показывает, что прирост эффективности вычислений зависит от задачи и ограничен сверху для любой задачи с $\alpha \neq 0$.

Таким образом, не для всякой задачи имеет смысл наращивание числа процессоров в вычислительной системе. Более того, если учесть время, необходимое для передачи данных между узлами вычислительной системы, то зависимость времени вычислений от числа узлов будет минимальна. Это накладывает ограничение на масштабируемость вычислительной системы, то есть означает, что с определенного момента добавление новых вычислительных элементов в систему будет увеличивать время расчёта задачи.

Однако в работе [10] Джона Густафсона на вычислительной системе из 1024 процессоров приведено параллельное решение трех больших задач, для которых доля последовательного кода α лежала в пределах от 0,5 до 0,8% и получены значения ускорения по сравнению с последовательным вариантом от 1016 до 1021 раз. Согласно закону Амдала, для данного числа процессоров и значений α ускорение не должно было превысить 200 раз. Густафсон пришел к выводу, что причина этого заключается в исходной предпосылке, лежащей в основе закона Амдала: увеличение числа процессоров не сопровождается увеличением объема решаемой задачи. Реальное же положение вещей существенно отличается от такой гипотезы.

Обычно, получая в свое распоряжение более мощную систему, пользователь не стремится сократить время вычислений, а сохраняя его практически неизменным, старается пропорционально мощности вычислительной системы увеличить объем вычислений, например, чтобы получить решение более сложных, имеющих практическое применение приложений решаемой задачи. И если оказывается, что наращивание общего объема касается главным образом распараллеливаемой части программы, то это ведет к сокращению величины α. Примером может служить искусственная нейронная

сеть. Если, например, доля последовательно выполняемого кода составляет 10% для 100 нейронов, то для 10 000 нейронов доля последовательно выполняемого кода снизится до 0,1%.

Таким образом, повышение ускорения обусловлено тем, что, оставаясь практически неизменной, последовательная часть в общем объеме увеличенной программы имеет уже меньший удельный вес. Чтобы оценить степень ускорения вычислений, когда объем последних увеличивается с ростом количества процессоров в системе (при постоянстве общего времени вычислений), Густафсон рекомендует использовать выражение, предложенное Е. Барсисом [11]:

$$S_p = \alpha + (1-\alpha)p\,.$$

Данное выражение известно как закон масштабируемого ускорения или закона Густафсона–Барсиса, из которого следует, что ускорение является линейной функцией числа процессоров, если рабочая нагрузка увеличивается так, чтобы поддерживать неизменным время.

Чтобы продемонстрировать, какие задачи можно решать с помощью параллельных вычислительных систем, можно рассмотреть несколько примеров. Пусть, например, выполняется умножение элементов двух векторов *A* и *B*, каждый из которых содержит *N* элементов, в результате получается вектор *C* (рис. 2) [2].

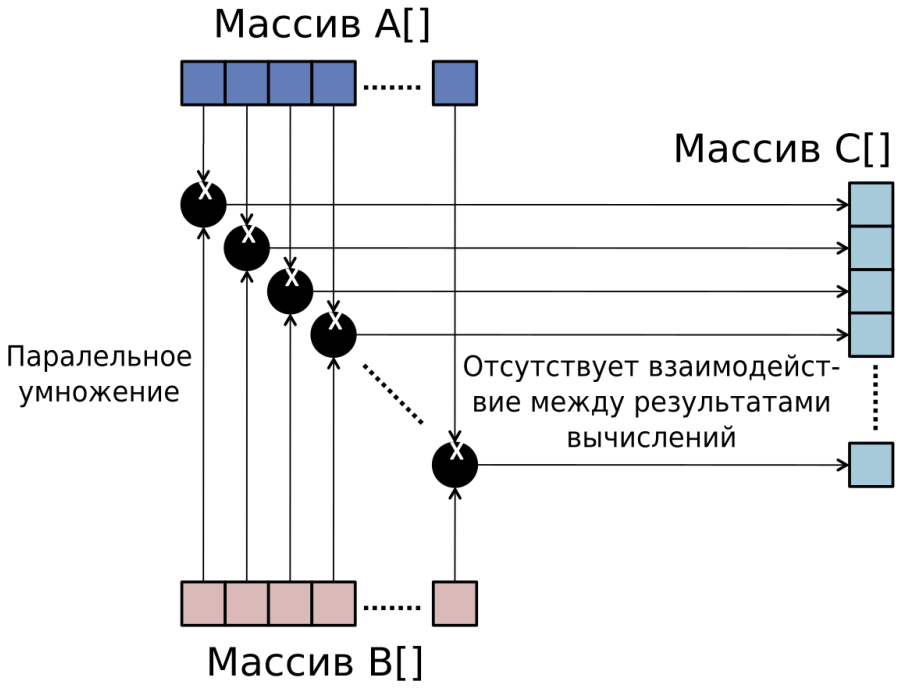

Рисунок 2. Поэлементное умножение векторов с распараллеливанием на уровне данных

Вычисление каждого элемента вектора *C* не зависит от любого другого его элемента. При распараллеливании этого кода можно использовать отдельный процессор для вычисления каждого элемента вектора *C*. Следовательно, в этом примере можно исходить из принципа параллелизма на уровне данных, поскольку одна и та же операция применяется ко всем элементам *A* и *B* для получения *C*, а максимальное количество использу-

емых вычислительных элементов ограничивается только их наличием и накладными расходами на пересылку исходных данных и получение результата.

На рис. 3 показан параллелизм на уровне задачи на примере фильтрации множества изображений с использованием БПФ. Каждое изображение в наборе может обрабатываться независимо друг от друга на отдельном процессоре.

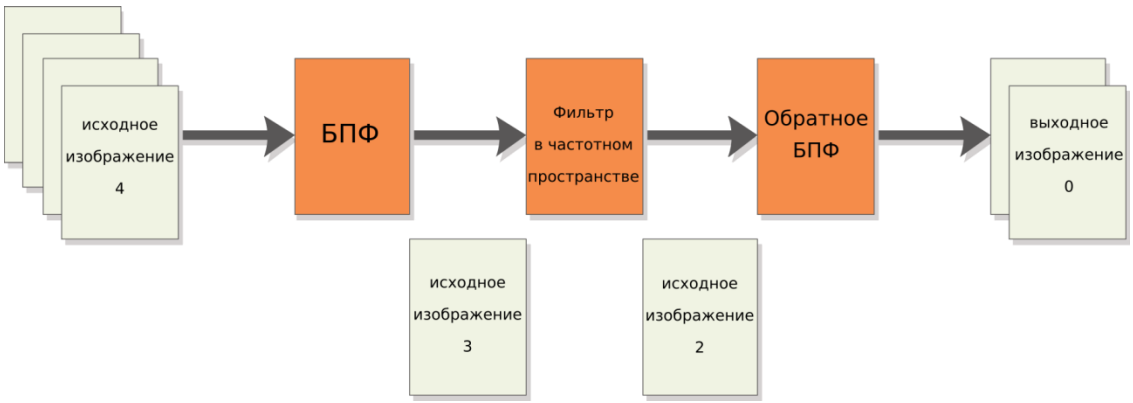

Рисунок 3. Фильтрация набора изображений с распараллеливанием на уровне задачи

Пусть, например, теперь нужно найти количество вхождений некоторого слова в текст (рис. 4).

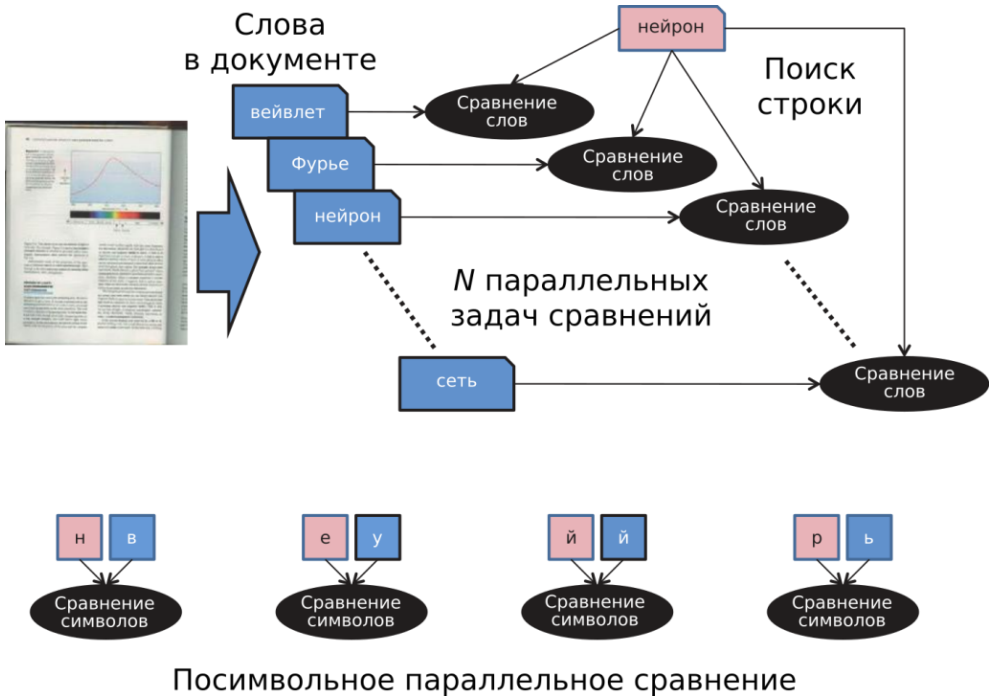

Рисунок 4. Поиск количества вхождений слова в текст

Предположим, что текст уже преобразован в набор из *N* слов, тогда можно сравнивать по очереди каждое слово с остальными *N-1* словами. Процесс сравнивания слова с множеством потенциально совпадающих с ним слов представляет собой *N* параллель-

ных выполняемых задач. Здесь существует еще и вложенный параллелизм при посимвольном сравнении двух слов. Этот пример демонстрирует параллелизм и на уровне данных (одна и та же операция посимвольного сравнения выполняется с несколькими элементами данных), и на уровне задачи (слово сравнивается со всеми другими словами одновременно). Как только количество совпадений определено, необходимо сложить их, чтобы получить общее количество вхождений. Это суммирование также может выполняться параллельно каскадным суммированием с временной логарифмической сложностью с помощью операции редукции (рис. 5) частичных сумм для нахождения конечного результата [2].

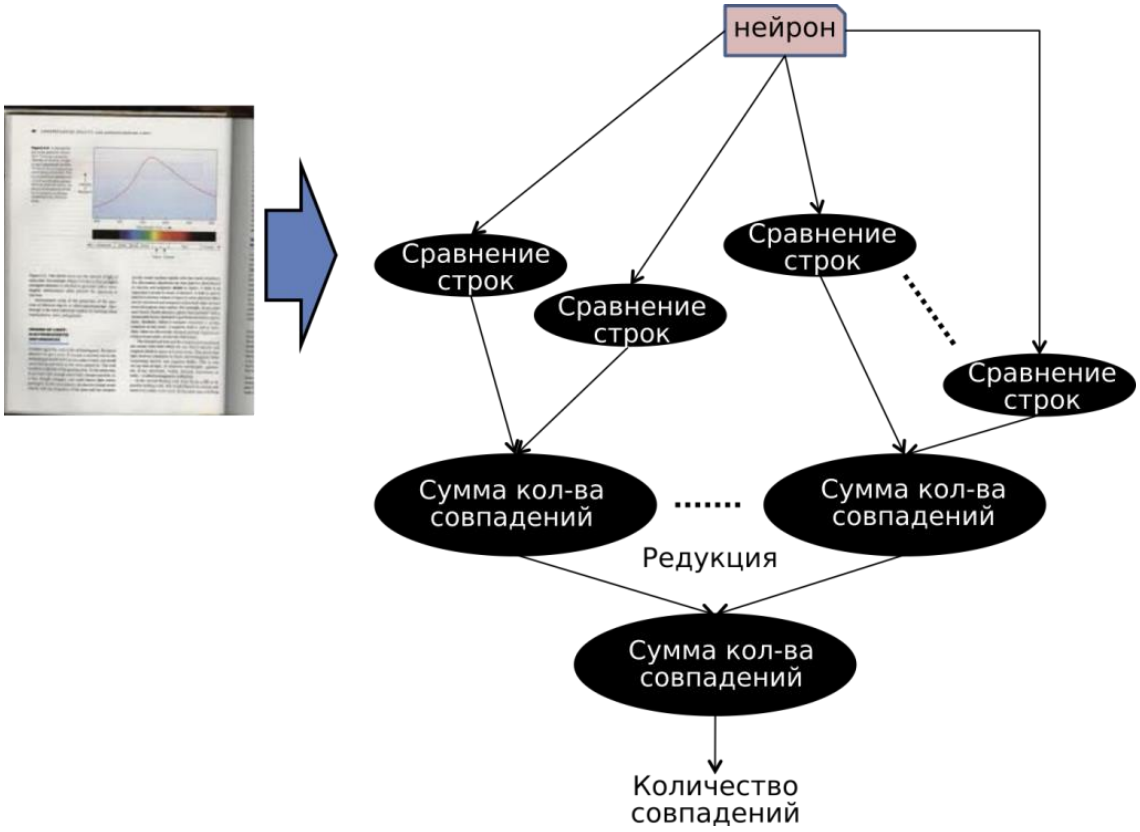

Рисунок 5. Каскадное суммирование

К понятию уровня параллельности тесно примыкает понятие гранулярности – это мера отношения объема вычислений, необходимых для параллельного решения задачи, к объему коммуникаций, необходимых для обмена сообщениями. Наиболее часто выделяют крупнозернистый, среднезернистый и мелкозернистый параллелизм.

- Крупнозернистый параллелизм имеет место тогда, когда каждое параллельное вычисление выполняется в большой степени независимо от остальных, причем требуется достаточно редкий обмен сообщениями между отдельными независимыми вычислениями. Единицами распараллеливания здесь являются большие программы, содержащие тысячи команд. Этот уровень параллелизма обычно обеспечивается операционной системой.

- При среднезернистом параллелизме единицами распараллеливания являются функции, включающие в себя сотни команд. Такой параллелизм может организовываться как программистом, так и компилятором.
- Мелкозернистый параллелизм имеет место тогда, когда каждое параллельное вычисление очень мало и содержит не более нескольких десятков команд. Обычно распараллеливаемыми единицами являются небольшие выражения или отдельные итерации циклов, с небольшими зависимостями по данным. Обычно термин «мелкозернистый параллелизм» означает наличие простых и быстрых любого вычислительных действий. Характерная особенность мелкозернистого параллелизма заключается в приблизительном равенстве интенсивности вычислений и обмена данными [13].

Эффективное параллельное программирование требует хорошего баланса между степенью гранулярности программ и величиной задержки, возникающей между разными гранулами из-за обмена данными. Если эта задержка минимальна, то максимальную производительность будет обеспечивать мелкозернистое разбиение программы. Это тот случай, когда нужно использовать методы параллелизма на уровне данных. Если коммуникационная задержка велика, предпочтительней крупнозернистое разбиение программы на уровне задачи.

Технологии параллельного программирования. Сущность параллелизма заключается в наличии двух или более действий, происходящих одновременно, что применительно к разработке программного обеспечения и означает наличие программно-аппаратной системы, выполняющей несколько задач независимо. Хотя возможно, что независимые задачи могут выполняться одновременно (то есть параллельно), на это не является обязательным требованием. В качестве примера можно привести программу для рисования, которая должна получать информацию об изображении от пользователя и выводить на дисплей сформированное изображение. Прием и обработка входных данных это задачи, которые могут быть независимыми от обновления изображения на мониторе. Хотя их и можно решать одновременно, но, как правило, в этом нет необходимости, тем более, если все они выполняются на одном процессоре. В этом случае программа или операционная система должны переключаться между задачами, по очереди предоставляя им время процессора. В настоящей работе рассматривается параллелизм, связанный с выполнением двух или более вычислений одновременно с явной целью повышения общей производительности.

Например, рассмотрим следующие вычисления :

$$A = B+C \quad (1)$$

$$D = E+G \quad (2)$$

$$R = A+D \quad (3)$$

Вычисления значений A и D в формулах (1) и (2) являются независимыми друг от друга, поскольку они не связаны общими данными. Однако выражение (3) зависит от выражений (1) и (2), поэтому не может вычисляться параллельно с (1) или (2). Параллельные программы обязательно должны иметь части, которые являются независимыми, за счет чего возможно их параллельное выполнение. Как правило, они оформляются в виде нескольких подпрограмм, называющихся потоками. Связь между потоками осуществляется через совместный доступ к памяти, в одном и том же адресном простран-

стве. Каждый поток имеет и свою локальную память, то есть свои локальные переменные, но все потоки имеют общие глобальные переменные. Основная программа содержит части, которые не могут выполняться параллельно с другими. Она запускается операционной системой, выполняет загрузку необходимых для запуска потоков системных и пользовательских ресурсов и создает ряд потоков. Для организации согласованного взаимодействия потоков через разделяемую память необходимы специальные конструкции синхронизации, такие как блокировки или семафоры [2].

Все многопроцессорные вычислительные системы в той или иной мере поддерживают разделяемую память. Однако поддержка полностью разделяемой между всеми процессорами памяти для многопроцессорных и многоядерных систем на аппаратном уровне является ограничивающим фактором, поскольку необходимое число соединений для передачи данных между процессорами растет по экспоненте с увеличением их количества. Поэтому часто используется менее жесткая модель разделяемой памяти, когда процессоры разбиваются на группы, и непосредственный обмен данными возможен только между процессорами одной группы, что позволяет достичь лучшей масштабируемости системы [14].

Кроме разделяемой памяти возможна организация явного взаимодействия через передачу сообщений, что требует совместных операций отправки и получения, которые должны выполняться каждым потоком. На практике такое взаимодействие реализуется с помощью библиотеки аппаратно-независимых подпрограмм для отправки и получения сообщений. Исторически сложилось так, что с 1980-х годов доступно множество библиотек для передачи сообщений между потоками, самой популярной из которых является MPI [15]. Однако ее реализации для различных операционных систем существенно различаются, что затрудняет разработку кроссплатформенных приложений.

Рассмотрим, например, случай, когда запускаются два потока, не использующие общие данные. Если операционная система имеет доступ к необходимым им ресурсам, их можно запускать одновременно или параллельно. Однако если во время выполнения одного потока он сгенерировал данные, которые могут потребоваться второму потоку, тогда возникает необходимость в некоторой форме синхронизации, а параллельное выполнение потоков через точку синхронизации становится невозможным, из чего следует, что при создании программного обеспечения для параллельных вычислительных систем важную роль играет обмен данными и синхронизация. Примерами совместного использования данных в параллельных программах являются и такие задачи, результат решения которых зависит от другой задачи, как например, в известных шаблонах производителя-потребителя или конвейера, а также когда промежуточные результаты объединяются вместе (например, как часть редукции, показанной на рис. 5). Лучше всего поддаются распараллеливанию части программы, лишенные зависимости от общих данных. Но на практике такие случаи встречаются не часто, и практически всегда и приходится использовать конструкции синхронизации, поддерживаемые CUDA и OpenCL.

Выводы. В настоящей работе дана характеристика современных гетерогенных вычислительных систем, кроме традиционных процессоров, содержащих, как правило, графические процессоры и другие вычислительные устройства. Даны краткие теоретические сведения и приведен обзор актуальных методов и технологий создания про-

грамм для параллельных вычислительных систем. На примерах показано, что применяя различные методы, можно максимально распараллелить задачу, чтобы добиться эффективного решения на многопроцессорных гетерогенных вычислительных системах.

Литература

1. Пантелеев А.Ю. Высокопроизводительные сопроцессоры для параллельной обработки данных в формате с плавающей точкой в системах цифровой обработки сигналов: автореф. дис. канд. техн. наук. – Москва, 2013. – 26 с.
2. Heterogeneous Computing with OpenCL 2.0. David R. Kaeli, Perhaad Mistry, Dana Schaa, Dong Ping Zhang. – Waltham: Morgan Kaufmann, 2015. – 330 p.
3. Пантелеев А.Ю. Цифровая обработка сигналов на современных графических процессорах // Цифровая обработка сигналов. – 2012. – № 3. – С. 68–75.
4. Кориков А.М, Симонов В.В. Гибридная архитектура параллельных вычислительных систем // Доклады Томского государственного университета систем управления и радиоэлектроники. – 2012. – Т.1. № 2(26). – С. 178–183.
5. http://www.nvidia.ru/object/cuda-parallel-computing-ru.html (дата обращения: 17.09.2017)
6. http://www.amd.com/ru-ru/innovations/software-technologies/firepro-graphics/stream (дата обращения: 17.09.2017)
7. https://www.khronos.org/opencl/ (дата обращения: 18.09.2017)
8. Almasi G. S., Gottlieb, A. Highly Parallel Computing. – Redwood City, CA: Benjamin Cummings, 1989.
9. Amdahl Gene M. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities // AFIPS Conference Proceedings. – 1967. N. 30. – P. 483–485.
10. Gustafson, J. L. Reevaluating Amdahl's Law // CACM. – 1988. N. 31(5). – P. 532–533.
11. Quinn M.J. Parallel Programming in C with MPI and OpenMP. – New York: NY: McGraw-Hill, 2004.
12. Линев А.В., Боголепов Д.К., Бастраков С.И. Технологии параллельного программирования для процессоров новых архитектур. М.: Изд-во Московского университета, 2010. – 160 с.
13. Орлов С. А., Цилькер Б. Я. Организация ЭВМ и систем: Учебник для вузов. 2-е изд. – СПб.: Питер, 2011. – 688 с.
14. Эндрюс, Г.Р. Основы многопоточного, параллельного и распределенного программирования. М.: «Вильямс», 2003.
15. Долинский М., Толкачев А. Обзор аппаратных и программных средств реализации параллельной обработки // Компоненты и Технологии. – 2004. – № 6. – С. 152–155.