УДК 004.421

Д.Ж. Ахмед-Заки, М.Б. Борисенко

НИИ Институт математики и механики при Казахском национальном университете им. аль-Фараби, Алматы, Казахстан; Darhan.Ahmed-Zaki@kaznu.kz, Mstislav.Borissenko@gmail.com

Разработка высокопроизводительных приложений с использованием гибридных технологий параллельных вычислений - MPI/OpenMP/Cuda*

В данной работе рассматривается создание гибридного высокопроизводительного приложения с использованием технологий параллельных вычислений MPI, OpenMP и CUDA. Полученные результаты тестирования приложения представлены и проанализированы, на основании чего описаны преимущества выбранной гибридной архитектуры.
Ключевые слова: высокопроизводительные приложения, гибридные технологии, параллельные вычисления, MPI, OpenMP, Cuda.

D.Zh. Ahmed-Zaki, M.B. Borissenko

Development of high-performance applications with the use of MPI/OpenMP/CUDA hybrid technologies of parallel computations

Creation of hybrid high-performance application with the use of MPI, OpenMP and Cuda technologies is considered in this paper. Results that were acquired from tests of the application are provided and analyzed, on the base of which advantages of the chosen hybrid architecture are described.
Key words: high-performance applications, hybrid technologies, parallel computing, MPI, OpenMP, Cuda.

Д.Ж. Ахмед-Заки, М.Б. Борисенко

MPI/OpenMP/Cuda деген параллельді есептеулердің гибрид технологияларының қолданымен жоғары өнімді қосымшалардың зерттеме

Бұл жұмыста MPI/OpenMP/Cuda деген параллельді есептеулер қолданымен гибрид жоғары өнімді қосымшаның жасауы көрсетілген. Қосымшаның сынақтамасынан алынған нәтижелер берілген және талдалған, осы талдау негізінде сайлаулы гибрид құрылымының артықтығылар мазмұндалған.
Түйін сөздер: жоғары өнімді қосымшалар, гибрид технологиялар, параллельді есептеулер, MPI, OpenMP, Cuda.

Цель работы и постановка задачи

Целью рассматриваемой работы являлось создание гибридной программы, выполняющей решение системы линейных алгебраических уравнений методом сопряженных градиентов, на языке программирования Java с использованием технологий MPI, OpenMP

*Работа выполнена при поддержке Комитета Науки МОН РК, грант № 1549 / ГФ3.

и CUDA для обеспечения использования как ядер центрального процессора компьютера, так и графического процессора видеокарты.

Поскольку программа была создана на языке программирования Java, при ее написании использовались реализации упомянутых технологий именно для этого языка - MPJ, JOMP и JCUDA.

Метод сопряженных градиентов - это численный метод решения систем линейных алгебраических уравнений. Ключевым понятием, используемым в этом методе, является понятие сопряженности векторов.

Векторы S_i и S_j называются сопряжёнными, если:

1. $S_i{*}H{*}S_j = 0 \ (i \neq j)$
2. $S_i{*}H{*}S_j \geq 0,$ где H - матрица Гессе

Решаемая система линейных уравнений должна быть представима в матричном виде: Ax = b, где матрица A - это действительная симметричная положительно определённая матрица. Алгоритм поиска решения системы линейных уравнений следующий:

1. Необходимо выбрать начальное приближение для вектора x - это может быть нулевой вектор.
2. $r^0 = b - Ax^0$.
3. $z^0 = r^0$.
4. $\alpha_k = \dfrac{(r^{k-1}, r^{k-1})}{(Az^{k-1}, z^{k-1})}$.
5. $x^k = x^{k-1} + \alpha_k z^{k-1}$.
6. $r^k = r^{k-1} + \alpha_k A r^{k-1}$.
7. $\beta_k = \dfrac{(r^k, r^k)}{(r^{k-1}, r^{k-1})}$.
8. $z^k = r^k + \beta_k z^{k-1}$.
9. Происходит возвращение к шагу 4, если не выполнен критерий останова. Для этого может использоваться значение модуля разности между решениями на соседних итерациях или значение относительной невязки $\dfrac{\|r^k\|}{\|b\|}$. Итерационный процесс может завершаться, когда это значение становится меньше заданной величины.

Основные результаты

На первом этапе работы была разработана программа генерации систем линейных уравнений заданного размера со случайными коэффициентами, удовлетворяющих необходимым требованиям, выполнение которых обязательно для применимости метода сопряженных градиентов. Для использования в этом программном комплексе были созданы классы Vector и Matrix, в качестве методов которых были реализованы все необходимые операции, включая умножение на матрицу, на вектор, на скаляр, сложение и вычитание.

Также на этом этапе была разработана последовательная программа, выполняющая решение сгенерированных систем линейных уравнений произвольного размера методом сопряженных градиентов.

Написанная программа была протестирована для различных систем размером от 10 до 100 уравнений. Поскольку матрица и векторы заполняются случайными числами, время исполнения программ может варьироваться в пределах 5-10%. В таблице 1 приведены усредненные значения на основе нескольких измерений.

Таблица 1. Время выполнения программы решения линейных уравнений методом сопряженных градиентов

Количество неизвестных	Время выполнения, *мс*
10	215
20	880
50	12800
100	91170

После написания последовательного варианта была создана параллельная программа на языке программирования Java, использующая технологии JOMP и MPJ для параллельного выполнения вычислений. При проектировании программы были рассмотрены различные варианты ее архитектуры и определена наиболее подходящая для решения данной задачи. В выбранной архитектуре учитываются особенности и используются сильные стороны каждой из технологий.

Стандарт MPI определяет, что каждый из потоков вычислений производится в отдельном процессе, что позволяет использовать различные адресные пространства и разделять их, перенося на другой компьютер и используя его память и вычислительные ресурсы. При этом обмен информацией между процессами происходит при помощи копирования и пересылки информации, что создает дополнительные расходы на буферизацию, обмен сообщениями и передачу информации, что не позволяет в полной мере использовать преимущества разделяемой между ядрами процессора памяти. С другой стороны, стандарт OpenMP определяет, что все потоки вычислений реализуются в нескольких потоках внутри одного общего процесса, что позволяет им использовать разделяемую память в пределах одного компьютера и использовать ссылки для передачи информации между потоками вместо ее полного копирования и переноса. [1]

MPJ может быть запущен в двух доступных конфигурациях - мультиядерной и кластерной. Мультиядерная конфигурация предназначена для использования на одном компьютере, и в этом случае ее использование эквивалентно использованию OMP, но в большинстве случаев может уступать по производительности, поскольку для передачи данных необходимо их копирование. Кластерная конфигурация предназначена для использования вычислительных ресурсов нескольких компьютеров при произведении расчетов, поскольку позволяет запускать отдельный процесс MPJ на каждом из используемых компьютеров.

Для запуска MPJ в кластерной конфигурации необходимо, чтобы на каждой из используемых машин был запущен процесс MPJ Daemon.

Первым вариантом реализации этого является регистрация в службах компьютера MPJ Daemon, который используется для запуска MPJ в кластерной конфигурации. Аль-

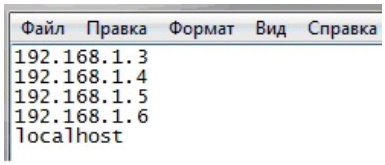

Рисунок 1. Файл машин для кластерной конфигурации

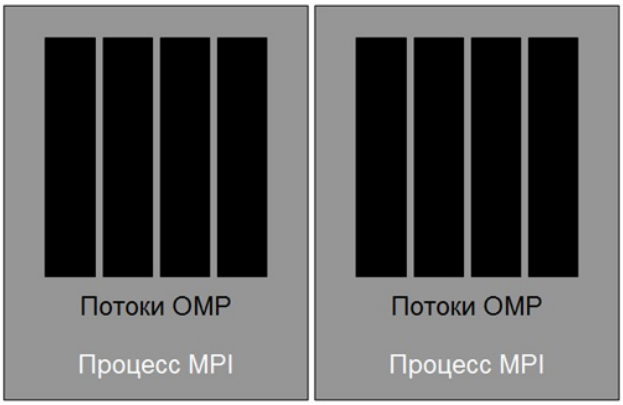

Рисунок 2. Архитектура программы с использованием технологий MPI и OMP

тернативным способом запуска программы в кластерной конфигурации является запуск MPJ Daemon перед запуском самой программы через bat-файл.

При запуске программы в кластерной конфигурации используется файл машин, в котором можно указывать, на каких компьютерах будут запускаться процессы MPJ. В этом файле все компьютеры, на которых будут запущены процессы вычислений, должны быть указаны по одному на каждой строке либо адресом, либо именем в сети. Последним должен быть указан текущий компьютер, что может быть осуществлено при помощи указания localhost. Подобный файл был создан и успешно использован. Содержимое файла приведено на Рисунке 1.

Поскольку созданная программа использует технологию MPI, она может быть запущена как в мультиядерной, так и в кластерной конфигурации. Выбор конфигурации может зависеть от доступного количества компьютеров и вычислительных ресурсов.

Распределение данных между процессами производилось при помощи MPJ, в то время как параллельная обработка данных внутри каждого из процессов выполнялась потоками JOMP.

Использованная архитектура программы показана на рисунке 2. При этом количество процессов MPI может быть выбрано любым в соответствии с количество используемых компьютеров и процессоров. Количество потоков OMP может соответствовать количеству ядер в процессорах.

Такая архитектура обеспечила масштабируемость программы, поскольку все потоки вычислений инкапсулированы в несколько независимых друг от друга и обменивающихся сообщениями процессов. Таким образом, стало возможным задействовать несколько компьютеров для выполнения вычислений, на каждом из которых для этого запускался процесс MPJ.

Подобная архитектура параллельной программы использует все преимущества каждой из использованных технологий. Использование MPI для передачи информации между процессами позволяет использовать вычислительные ресурсы большого числа компьютеров одновременно, в то время как использование OpenMP для многопоточной обработки информации позволяет максимально эффективно использовать преимущества разделяемой памяти, уменьшая время, затрачиваемое на синхронизацию и пересылку информации.

Написанная параллельная программа была протестирована для различных систем размером от 2 до 100 уравнений с использованием 4 потоков вычислений на 4х ядерном компьютере. Результаты тестирования представлены в таблице 2.

Таблица 2. Время выполнения параллельной гибридной программы с использованием технологий MPJ и JOMP

Количество неизвестных	Время выполнения, *мс*
10	110
20	285
50	3550
100	23200

Рассмотренный вариант программы максимально эффективно использует вычислительные ресурсы центральных процессоров одного или нескольких компьютеров. Тем не менее, большинство современных рабочих станций снабжено видеокартами с мощными графическими процессорами. Вычислительные ресурсы видеокарт также можно использовать для проведения необходимых расчетов с использованием технологии CUDA.

JCUDA Bindings - привязка библиотек CUDA для языка программирования Java.

Написанная параллельная программа была дополнительно модифицирована и усовершенствована для реализации использования вычислительных ресурсов видеокарты компьютера.

Функция ядра CUDA была написана самостоятельно на языке программирования C++. Поскольку формат данных, ожидаемых функцией ядра, написанной на C++, должен соответствовать формату данных, передаваемых из программы, написанной на Java, оказалось необходимо использовать особые структуры данных C++. Для реализации структуры данных, аналогичной двумерному массиву языка Java был создан массив указателей на указатели на языке C++.

Написанная функция ядра поддерживает возможность умножения нескольких строк матрицы одновременно на один и тот же вектор. В созданной функции ядра каждый из потоков выполняет умножение элементов матрицы на элементы вектора, после чего сложение происходит методом редукции. Все потоки, выполняющие вычисления, делятся на две равные группы. Потоки, выполнявшие вычисления со значениями первой группы, прибавляют к своим элементам результаты из второй группы. После этого операция повторяется рекурсивно по отношению к первой половине значений на каждом этапе. В итоге единственное получившееся значение является результатом проведения операции редукции и записывается в разделяемую переменную-результат. Это позволяет на каждом этапе производить суммирование параллельно нескольким потокам.

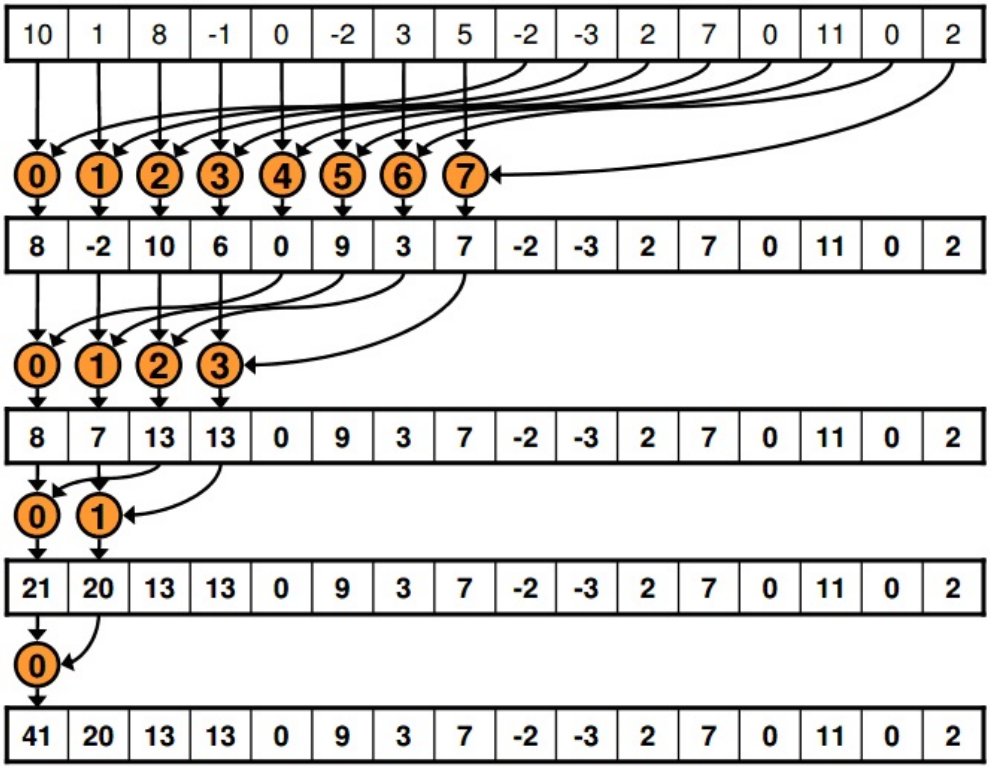

Рисунок 3. Использованный процесс редукции в функции ядра CUDA[2]

Схема редукции показана на Рисунке 3, а время выполнения параллельной гибридной программы с использованием технологий MPJ, JOMP и JCUDA приведено в таблице 3.

Таблица 3. Время выполнения параллельной гибридной программы с использованием технологий MPJ, JOMP и JCUDA

Количество неизвестных	Время выполнения, *мс*
10	270
20	420
50	1050
100	4250

На основании полученных данных были рассчитаны полученное среднее ускорение и средняя эффективность параллельных версий программы при решении систем уравнений с различным числом неизвестных. Результаты приведены в Таблицах 4 и 5.

Полученное сверхлинейное ускорение при использовании технологии CUDA объясняется высокой степенью оптимизации функции ядра и большим количеством ядер графического процессора. При этом инициализация графического процессора и получение контекста для выполняемой программы занимает в среднем от 100 до 200 мс, вследствие чего при решении систем уравнений незначительного объема большая часть времени выполнения программы тратится на инициализацию, что приводит к низкому ускорению и малой эффективности. При росте объема системы и числа неизвестных большее

число ядер графического процессора используется программой и выполняет расчеты, вследствие чего резко растет ускорение и эффективность.

Таблица 4. Полученное ускорение для параллельных версий программы при использовании 4 потоков на 4х ядерном процессоре

Количество неизвестных	JOMP	MPJ+JOMP	MPJ+JOMP+JCUDA
10	1.95	1.08	0.8
20	3.09	2.32	2.1
50	3.75	3.87	12.19
100	3.85	3.93	21.45

Таблица 5. Полученная эффективность параллельных версий программы при использовании 4 потоков на 4х ядерном процессоре

Количество неизвестных	JOMP	MPJ+JOMP	MPJ+JOMP+JCUDA
10	0.49	0.27	0.2
20	0.77	0.58	0.53
50	0.93	0.97	3.05
100	0.96	0.98	5.37

Полученные результаты для гибридной программы с использованием технологий MPI и OpenMP согласуются с результатами, полученными британскими специалистами при анализе производительности гибридных программ на многоядерных кластерах: использование только одной технологии может быть более эффективным при проведении небольшого количества вычислений на одном вычислительном узле с небольшим количеством ядер.

В то же время гибридная реализация программы более эффективна при работе на большом количестве узлов, поскольку намного лучше масштабируется и имеет возможность использовать ядра процессоров сразу нескольких компьютеров [3]. Кроме того, использование оптимизированных вычислений с использованием графического процессора позволяет значительно ускорить расчеты на компьютерах, оснащенных видеокартой, поддерживающей технологию CUDA.

Список литературы

[1] *Douglas Eadline* Comparing MPI and OpenMP. – 2013. – (http://www.clusterconnection.com)

[2] *Mark Harris* Optimizing Parallel Reduction in CUDA. – 2013. – (http://developer.download.nvidia.com)

[3] *Martin J. Chorley, David W. Walker* Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters // Journal of Computational Science. – 2011. – V.1. – Issue 3. – pp. 168–174.

Поступила в редакцию 13 сентября 2013 года