Конвейерная модель представления параллельных программ

М.А. Кривов, М.Н. Притула

Московский государственный университет им. М.В. Ломоносова

Введение

В связи с широким распространением многоядерных архитектур всё более и более актуальной становится проблема разработки подходов, позволяющих создавать эффективные параллельные реализации обычных неспециализированных алгоритмов. Данные подходы должны быть достаточно гибкими и удобными, так как будут использованы разработчиками, которые могут и не иметь достаточной подготовки в области параллельных вычислений. С другой стороны, в настоящее время набирают популярность гибридные архитектуры, в которых есть сразу несколько вычислителей с различными интерфейсами. В качестве примера таких архитектур можно привести обычный многоядерный персональный компьютер, оборудованный графическим ускорителем от NVidia, поддерживающим технологию CUDA. Также в 2010 году корпорация Intel планирует выпустить архитектуру Larrabee, представляющую собой карту расширения примерно с 40 x86-совместимыми ядрами. В результате этого также становятся востребованными подходы, позволяющие унифицировать взаимодействие с интерфейсами различных вычислительных устройств.

Существует множество различных решений, предлагающих решения вышеозвученных проблем. В основном они реализованы как расширения для существующих языков программирования[1] или как библиотеки готовых примитивов и паттернов взаимодействия[2]. Основным их недостатком является решение только одной из двух названных проблем, в результате чего подход получается либо достаточно сложным и узкоспециализированным, либо ориентированным на конкретную платформу.

Описание проекта

Целью настоящей работы является создание библиотеки для разработки параллельных приложений, использующей только один примитив — конвейер. Данный примитив является высокоуровневой конструкцией и предполагает независимость между составляющими его звеньями, в результате чего становится возможным создание модели, способной интегрировать различные API и при этом оставаться достаточно гибкой и естественной.

Конвейерная модель программ также обладает рядом достоинств. В разрабатываемой библиотеке получили развитие следующие из них:

1. Возможность автоматического сбора статистики. Так как каждое звено конвейера производит однотипную обработку данных, то, зная статистику за предыдущие запуски, можно предсказать время обработки конкретной порции данных. Данная информация может быть использована для динамической балансировки нагрузки на доступные вычислители, а также для нахождения критического пути программы.
2. Использование гибкой компонентной модели. В предлагаемой реализации конвейер является произвольным соединением независимых блоков и предоставляет возможность динамического перестроения. Это позволяет переиспользовать существующие блоки, а также упрощает процесс изменения параметров работающей программы (так как для этого достаточно только пересоздать существующий блок). Более того, изменение структуры конвейера может быть произведено пользователем через специальные графические утилиты, динамически подключающиеся к программе и изменяющие его параметры.

---

1 Например таких как OpenMP, NVidia CUDA, AMD Stream, Intel Ct, Microsoft Parallel Extensions.

2 Таких как Intel TBB, Microsoft CCR, MPI.

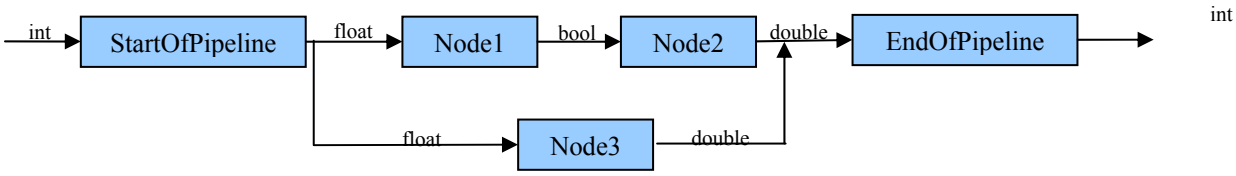
3. Возможность обработки данных на разных вычислительных устройствах. Каждое звено конвейера независимо от других, в результате этого разработчик может сделать дублирующие реализации конкретных звеньев с использованием различных API, что позволит системе времени выполнения использовать все доступные устройства.

Данный проект разрабатывается как независимая библиотека ttgLib для языка Microsoft Visual C++, последняя версия и более детальная информация о которой доступны на портале проекта http://ttglib.org. Также данный проект получил поддержку в рамках программы «У.М.Н.И.К.».

Из недостатков предлагаемого подхода стоит отметить невозможность эффективного представления некоторых задач в конвейерной форме. К таким проблемам, в частности, относятся задачи, имеющие рекурсивное решение или решение с множеством зависимостей. Для подобных задач планируется выпуск отдельной версии разрабатываемой библиотеки, интегрирующейся в библиотеку Intel Threading Building Blocks. Это позволит использовать предлагаемую конвейерную модель совместно с другими многопоточными примитивами, более удобными для решения данного класса задач.

Описание конвейерной модели

В разрабатываемой библиотеке конвейер реализован как ориентированный граф (возможно непланарный), вершины которого соответствуют звеньям, а рёбра — потокам данных заданного типа. Каждое звено может получить порцию данных, обработать её, и послать следующим звеньям. В этом графе также выделяются два особых звена — начало и конец конвейера, в которые данные могут быть посланы или, соответственно, получены непосредственно пользователем. Таким образом конвейер может, например, иметь следующий вид:



В предлагаемой модели вводится понятие итерации конвейера — процесса обработки всех данных, посланных на вход началу конвейера. Для достижения большей гибкости вводятся также события конвейера (такие как начало и конец итерации), на которые пользователь может привязать свои обработчики. С помощью них становится возможно реализовать работу с внешними сущностями (например, такими как файлы или OpenGL устройство), требующими инициализации перед первым использованием и закрытия после последнего.

Одной из отличительных особенностей от аналогов является возможность создания произвольных петель. Например, звено может отправить данные не только последующим звеньям, но и произвольному звену, удовлетворяющему некоторому критерию, что позволяет реализовывать задачи, для решения которых требуется несложная рекурсия. Для уменьшения количества звеньев, необходимый для решения конкретных задач, было решено добавить возможность переупаковки данных на уровне звена. С помощью данного механизма звено может начать обработку данных не сразу, а как только накопится нужное их количество или произойдёт определённое событие. После этого оно может выбрать нужное количество данных, переупаковать их в новый формат и отправить на обработку.

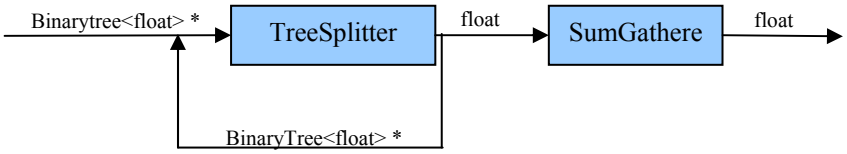
Как уже отмечалось ранее, в данной модели предполагается, что процесс обработки данных для каждого звена однотипен, что позволяет предсказать время их обработки. Для реализации данного принципа каждое звено может может возвращать алгоритмическую сложность обработки пришедших данных в произвольных единицах. После

этого система времени выполнения, зная отношение временной сложности к абстрактной для предыдущих итераций, с помощью интерполяции сможет получить ожидаемое время обработки. Как показали тесты, для большинства задач достаточно линейной интерполяции, в результате чего погрешность предсказания не превышала 3%.

Решение тестовых задач и сравнение с аналогами

С помощью описанной модели были решены следующие тестовые задачи, являющиеся модифицированными примерами из SDK для библиотеки Intel Threading Building Blocks:

- Суммирование элементов двоичного дерева. В этой задаче требуется для заданного двоичного дерева найти сумму всех элементов, хранящихся в узлах дерева. Чтобы упростить пример, считается известным количество узлов для каждого поддерева.
  Решением данной задачи является следующий конвейер из двух звеньев:

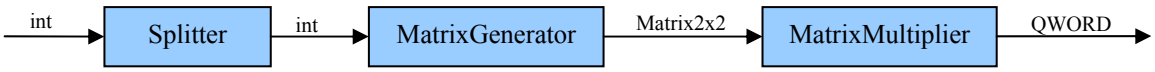


  Первое звено для каждого приходящего дерева осуществляет следующую проверку: если размер дерева меньше некоторого порога, то сумма его элементов считается рекурсивно и отправляется дальше. Иначе оно разбивается на два поддерева, которые отправляются этому же звену, а значение корневого элемента отправляется следующему звену.
  Второе звено суммирует все приходящие значения и по событию завершения итерации конвейера отправляет накопленную сумму дальше на выход конвейера.
- Обработка файла. В данной задаче требуется применить однотипную обработку к содержимому файла. В качестве обработки было решено вычислять хэш-функцию от содержащихся в файле данных. Для решения этой задачи был построен конвейер следующего вида:

  Первое звено производит непосредственную загрузку данных из файла и отправляет их следующему звену как набор массивов символов фиксированного размера. Второе звено применяет хэш-функцию к приходящим данным и отправляет полученные значения третьему звену, производящему их свёртку. При наступлении события завершения итерации последнее звено отправляет полученное значение хэш-функции на выход конвейера.
- Нахождение чисел Фибоначчи. Для нахождения *n*-ого числа Фибоначчи использовалось возведение в степень матрицы специального вида:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}$$

  После нахождения данной степени элемент a_{11} и будет искомым числом. Для решения этой задачи был построен конвейер следующего вида:



На вход первому звену подаётся номер искомого числа, после чего он бьётся на несколько слагаемых, количество которых зависит от количества ядер и которые отсылаются дальше по конвейеру. Звено *MatrixGenerator* возводит матрицу указанного вида

в степень, равную пришедшему числу, и отсылает её последнему звену, которое серийно перемножает все пришедшие матрицы и в конце итерации отправляет полученный результат на выход.

Для данных задач были также адаптированы решения с использованием Intel TBB и проведено сравнение всех полученных реализаций на компьютерах с различными архитектурами CPU. Результаты тестирования приведены в следующей таблице:

Метрика	Суммирование элементов дерева			Обработка файла			Вычисление чисел Фибоначчи		
	Serial	ttgLib	TBB	Serial	ttgLib	TBB	Serial	ttgLib	TBB
Количество строк кода	18	61	65	28	75	93	25	64	57
Время работы (Intel Core 2 Quad Q9400, 4 cores)	0.12 с (100%)	0.041 с (298%)	0.039 с (312%)	1.47 с (100%)	0.37 с (393%)	0.47 с (309%)	0.38 с (100%)	0.094 с (401%)	0.095 с (401%)
Время работы (Intel Pentium D 945, 2 cores)	0.21 с (100%)	0.11 с (192%)	0.1 с (196%)	3.09 с (100%)	1.57 с (196%)	1.56 с (198%)	0.48 с (100%)	0.29 с (164%)	0.30 с (159%)
Время работы (Intel Atom N270, 1 core with HT)	0.3 с (100%)	0.21 с (142%)	0.22 с (136%)	4.6 с (100%)	2.98 с (154%)	2.94 с (156%)	1.37 с (100%)	1.15 с (119%)	1.17 с (117%)

Для значений временных метрик в скобках также указано ускорение по сравнению с последовательной реализацией. Стоит отметить, все программы были написаны с использованием одного стиля программирования, поэтому количество строк кода отражает сложность реализации конкретной задачи с использованием соответствующего подхода.

Результаты тестирования показывают, что решения с использованием предлагаемой библиотеки незначительно уступают по производительности решениям, использующим Intel TBB, а в некоторых случаях даже обходят их. Стоит ещё раз отметить, что все рассмотренные задачи являются примерами из Intel TBB SDK, и их реализации используют различные примитивы, в то время как в разрабатываемой библиотеке используется только конвейер.

Заключение

В данной работе была предложена гибкая модель конвейера для создания параллельных программ для гибридных архитектур, позволяющая представить в конвейерном виде даже задачи, не имеющие явной конвейерной структуры. С помощью разработанного прототипа библиотеки был решён ряд тестовых задач и произведено сравнение полученных реализаций с решениями, созданными с помощью многопоточной библиотеки Intel Threading Building Blocks.

Литература

1. Воеводин В.В., Воеводин В.В. Параллельные вычисления. Серия "Научное издание". Издательство «BHV», СПб, 2002.
2. Кривов М.А., Притула М.Н. Система параллельной конвейерной обработки данных. Материалы докладов XVI Международной конференции студентов, аспирантов и молодых ученых «Ломоносов». Издательство «МАКС Пресс», Москва, 2009 .
3. Intel Threading Building Blocks Tutorial, электронный ресурс http://www.threadingbuildingblocks.org/documentation.php
4. NVidia CUDA Programming Guide, электронный ресурс http://www.nvidia.com/object/cuda_develop.html.