

Intel® Concurrent Collections for C++ - a model for parallel programming

Nikolay Kurtov
Intel Corporation
email:
nikolay.kurtov@intel.com

Ilya Cherny
Intel Corporation
email:
ilya.s.cherny@intel.com

William Youngs
Intel Corporation
email:
william.youngs@intel.com

Kath Knobe
Intel Corporation
email:
kath.knobe@intel.com

Geoff Lowney
Intel Corporation
email:
geoff.lowney@intel.com

Shin Lee
Intel Corporation
email:
shin.lee@intel.com

Stephen Rose
Intel Corporation
email:
stephan.rose@intel.com

Leo Treggiari
Intel Corporation
email:
leo.treggiari@intel.com

Judy Ward
Intel Corporation
email:
judy.ward@intel.com

Abstract

There is a challenge to grow the community of developers who can successfully exploit multi-core because application developers, who are experts in their domains, are not necessarily parallel or performance experts. Application providers have limited resources to develop and maintain multiple target-specific variants of their source code. Parallel programming is just too error-prone and time-consuming for wide scale adoption

Intel® Concurrent Collections is a simple yet powerful parallel programming model which separates the expression of all potential parallelism in an application both from the serial computations and also from all the target-specific details. This model raises the level of a programming language just enough to avoid typical parallelization issues and requires domain experts to define a semantically correct algorithm only.

A program in this model consists of an abstract parallel algorithm definition, high level primitive operations and data structures implemented in a serial language. All target-specific issues are solved by the runtime system. This is a very general approach of defining a parallel algorithm and it makes easy expressing any kind of parallelism. Intel® Concurrent Collections applications are not target-specific and do not have to be rewritten when they are ported to another platform.

Intel® Concurrent Collections for C++ is implemented as a C++ library and published on whatif.intel.com site (<http://softwarecommunity.intel.com/articles/eng/3862.htm>). Implementation includes a translator from textual abstract parallel algorithm definition to classes declaration.

Two standard performance benchmarks were modified to deploy Intel® Concurrent Collections for C++. Performance analysis demonstrated viability of the model and its implementation showing excellent scalability in some cases. Also some future areas for improvement of the implementation were identified.

Keywords: Parallel programming; C++ library; multi-core; multi-threading.

Модель параллельного программирования Intel® Concurrent Collections для C++

Nikolay Kurtov
Intel Corporation
email:
nikolay.kurtov@intel.com

Ilya Cherny
Intel Corporation
email:
ilya.s.cherny@intel.com

William Youngs
Intel Corporation
email:
william.youngs@intel.com

Kath Knobe
Intel Corporation
email:
kath.knobe@intel.com

Geoff Lowney
Intel Corporation
email:
geoff.lowney@intel.com

Shin Lee
Intel Corporation
email:
shin.lee@intel.com

Stephen Rose
Intel Corporation
email:
stephan.rose@intel.com

Leo Treggiari
Intel Corporation
email:
leo.treggiari@intel.com

Judy Ward
Intel Corporation
email:
judy.ward@intel.com

Тезисы

Интересна задача увеличения сообщества программистов, успешно использующих возможности многоядерных архитектур. Разработчики приложений ограничены в ресурсах на разработку и поддержание различных платоформенно-зависимых вариантов исходных кодов. Параллельное программирование слишком сложно и подвержено ошибкам, что замедляет его широкое распространение.

Intel® Concurrent Collections – простая, но очень мощная модель параллельного программирования, отделяющая весь потенциальный параллелизм в приложении и от последовательных вычислений, и от платоформенно-зависимых деталей. Эта модель поднимает уровень языка программирования ровно настолько, чтобы избежать типичных проблем параллелизации, и позволяет экспертам предметной области лишь определять семантически корректный алгоритм.

Программа, написанная в модели Intel® Concurrent Collections, состоит из абстрактного определения параллельного алгоритма, высокоуровневых операций и структур данных, реализованных в последовательном языке. Все платоформенно-зависимые вопросы решаются системой исполнения. Это очень общий подход к определению параллельного алгоритма, делающий лёгким выражение любого типа параллелизма. Приложения в этой модели не являются платоформенно-зависимыми и не нуждаются в переписывании при переносе на другую платформу.

Intel® Concurrent Collections для C++ реализована как C++ библиотека и опубликована на сайте [whatif.intel.com](http://softwarecommunity.intel.com/articles/eng/3862.htm) (<http://softwarecommunity.intel.com/articles/eng/3862.htm>). Реализация включает в себя транслятор из текстового абстрактного описания алгоритма в определение классов.

Два приложения для анализа производительности системы были реализованы с помощью данной модели и подтвердили её жизнеспособность, в некоторых случаях продемонстрировали хорошую масштабируемость, а также определили области для дальнейшего развития модели.

Keywords: *Параллельное программирование; библиотека C++; многоядерность; многопоточность.*

1. Вступление

Многоядерные архитектуры распространяются всё шире, а для полного использования предоставляемых возможностей всё важнее становится параллельное программирование. Само по себе параллельное программирование очень сложно, подвержено ошибкам и зависит от лежащей в основе архитектуры. Сейчас очень интересна задача увеличения сообщества программистов, успешно использующих возможности многоядерных архитектур. Одним из способов решения данной задачи является разработка средств, позволяющих избежать низкоуровневого программирования многопоточности.

Одним из таких средств является проект Intel® Concurrent Collections for C/C++. Это очень простая и мощная модель параллельного программирования, отделяющая выражение всего потенциального параллелизма в приложении и от последовательных вычислений, и от архитектурно-зависимых деталей.

2. Модель Intel® Concurrent Collections

В чём основная сложность параллельного программирования? Разработчик должен заботиться о выражении параллелизма, что часто приводит к тому, что разработчик уделяет много внимание не самому приложению, а работе с параллелизмом.

Другая причина, затрудняющая параллельное программирование, – оно встроено в последовательные языки программирования, что влечет за собой явное упорядочивание операций и перезапись данных.

Intel® Concurrent Collections поднимает уровень языка ровно настолько, чтобы избежать упомянутых проблем. Вводится не расширение языка, а интерфейс для описания задачи. Это позволяет решать задачу специалисту предметной области, который не является специалистом в области параллельного программирования. Таким образом, от разработчика требуется написать семантически корректную последовательную программу в рамках накладываемых Intel® Concurrent Collections ограничений. С него снимаются многие низкоуровневые вопросы:

- оптимизация под конкретную архитектуру
- написание платформенно-зависимого кода для работы с потоками
- балансировка загрузки потоков

- распределение задач между процессорами

Решением этих задач может заниматься другой человек, специалист по оптимизации, незнакомый с предметной областью, или статический анализ или анализ времени исполнения.

3. Intel® Concurrent Collections и другие модели параллельного программирования

Подход Intel® Concurrent Collections значительно отличается от других существующих моделей параллельного программирования.

При использовании Cilk[1] программист явным образом указывает вызовы функций, которые могут исполняться параллельно, а также о синхронизации при помощи барьеров, но при этом текст программы при удалении из него ключевых слов Cilk является корректной C/C++ программой.

OpenMP[2] очень хорошо подходит для задач, обладающих параллелизмом по данным. Базовые возможности OpenMP очень просты в освоении, что и делает эту модель очень популярной. Использование OpenMP требует поддержки компилятора.

Intel® TBB[3] предлагает высокоуровневую платформеннонезависимую абстракцию для параллельного программирования, а также множество базовых параллельных алгоритмов, которые могут значительно сократить объём кода и увеличить его качество и надёжность. Использование Intel® TBB требует хороших навыков применения шаблонов языка C++.

Во всех вышеперечисленных моделях программисте явно заботится о выражении параллелизма и имеет дело с императивным описанием алгоритма. Такой подход позволяет иметь больший контроль над производительностью программы, но заставляет разработчика думать о многих низкоуровневых вопросах.

Подход Intel® Concurrent Collections отличается тем, что предлагаемое естественное функциональное представление алгоритма требует лишь явного указания зависимостей в алгоритме, что делает возможным выявление всего потенциального параллелизма алгоритма. Также такое описание алгоритма является значительно более общим, что позволяет эффективно выражать любой тип параллелизма.

С другой стороны, скрывая все низкоуровневые вопросы параллелизма от разработчика, Intel® Concurrent Collections также

лишает его возможности значительно влиять на производительность программы.

4. Основные понятия

Основные понятия будут рассмотрены на примере простого приложения Blackscholes, решающего дифференциальное уравнение Блэка-Шоулза для множества наборов входных параметров. Данная программа представлена в наборе приложений для тестирования производительности системы PARSEC[4].

Способ описания задачи в модели Intel® Concurrent Collections представляет собой взгляд разработчика на решаемую задачу, который дополнен ровно настолько, чтобы сделать возможным исполнение данного представления.

В описании алгоритма разработчик отвечает на следующие вопросы:

- Какие выделены высокоуровневые операции?
- Какие данные используются?
- Какие присутствуют отношения производителя/потребителя?
- Какие данные являются входными, какие выходными?

Графическое представление алгоритма приложения Blackscholes представлено на рисунке 1.

Таким образом выделены два типа сущностей: высокоуровневые операции, называемые шаг (Step), и элементы данных (Item). В примере Blackscholes шаг называется «Решение», элементы данных называются «Параметры» и «Ответ». Прямые стрелки обозначают отношения производителя и потребителя, а фигурными стрелками обозначены входные и выходные данные.

Для уточнения этого описания разработчику требуется ответить также на следующие вопросы:

- Как различаются экземпляры шагов и элементы данных?
- Каковы наборы шагов и элементов данных?
- Кем определяются эти наборы?

Для ответа на эти вопросы к экземплярам шагов и элементов данных добавляются уникальные идентификаторы (Tag), имеющие произвольное число атрибутов произвольного типа.

Набор однотипных элементов данных

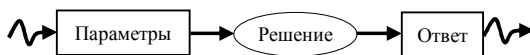


Рисунок 1. Графическое представление алгоритма Blackscholes

образует пространство элементов данных (Item Space), набор шагов образует пространство шагов (Step Space) и набор идентификаторов образует пространство идентификаторов (Tag Space).

На рисунке 2 изображен алгоритм приложения Blackscholes в графическом представлении Intel® Concurrent Collections. Пространства шагов обозначаются овалом, идентификаторов – треугольником, а пространства элементов данных обозначаются прямоугольниками.

Также вводится понятие отношения между сущностями. Сущности могут находиться в отношении подписывания (Prescriptive relations), что на графическом представлении обозначается пунктирной стрелкой, или в отношении производителя и потребителя (Producer and consumer relations) – сплошной стрелка.

В отношении подписывания могут находиться только пространство идентификаторов и пространство шагов. При этом для каждого идентификатора из первого пространства будет создан и исполнен шаг второго пространства, помеченный этим идентификатором. При этом нет ограничений на число пространств шагов, подписанных одним и тем же пространством идентификаторов.

В отношении потребителя состоят пространство элементов данных и пространство шагов, при этом шаг может использовать данные из указанного пространства элементов данных.

Отношение производителя – отношение между пространством шагов и пространством элементов данных или идентификаторов. Это означает, что шаг может производить элементы данных указанного типа или идентификаторы, что делает доступными для исполнения новые шаги.

Как эта модель работает для примера Blackscholes? До начала вычислений программист добавляет в пространство элементов данных все необходимые наборы параметров и для каждого из них создаёт соответствующий идентификатор. После запуска вычислений для каждого созданного идентификатора выполняется высокоуровневая операция, использующая данные из пространства «Параметры» и

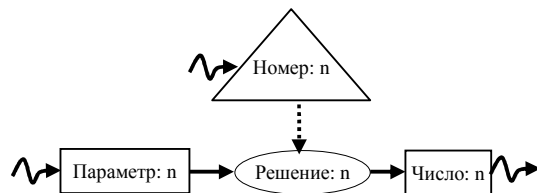


Рисунок 2. Графическое описание алгоритма Blackscholes

производящая элементы данных «Ответ». После выполнения всех вычислений результирующие данные становятся доступны программисту для дальнейшей работы с ними.

В таком описании алгоритма разработчику нигде не приходится заботиться о параллелизме приложения, тем не менее, он должен соблюдать некоторые естественные ограничения. Предполагается, что шаги выполняются атомарно, шаги могут взаимодействовать с глобальными данными только путём добавления и получения данных из пространств элементов. Запрещается модифицировать глобальные данные напрямую, это может повлиять на результат выполнения других шагов.

Таким образом, вводится достаточно простая модель параллельного программирования, освобождающая специалиста предметной области от низкоуровневых вопросов параллельного программирования. Модель является универсальной и может быть применена практически к любому языку и архитектуре: как к архитектуре с общей памятью, так и к архитектуре с распределённой памятью.

5. Текущая реализация для C++

На данный момент имеется реализация модели Intel® Concurrent Collections для C++, использующая библиотеку для параллельного программирования Intel® Threading Building Blocks, выполненная в виде шаблонной библиотеки. Высокоуровневое описание алгоритма производится не в графическом, а в текстовом виде. Правила текстового описания очень просты, на рисунке 3 приведены текстовые эквиваленты графическим обозначениям.

Для Blackscholes текстовое описание алгоритма представлено на рисунке 4. Входные и выходные данные обозначены как производимые или потребляемые окружением. В данном случае окружение – это C++ программа, производящая начальную подготовку данных и их обработку







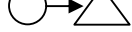
Элемент данных		[]
Шаг		()
Идентификатор		< >
Потребитель		[] → ()
Производитель		() → []
		() → < >
Подписка		< > :: ()

Рисунок 3. Соответствие между графическим и текстовым представлениями

```
//Объявления
[Data* parameters];
[double results];
<number: int n>;
//Отношения подписки
<number> :: (Solve);
//Отношения производителя/потребителя
env -> <number>;
env -> [parameters];
[parameters] -> (Solve);
(Solve) -> [results] -> env;
```

Рисунок 4. Текстовое представление алгоритма Blackscholes

после завершения исполнения описанного графом алгоритма. В описании элементов данных также необходимо указать тип данных, которые содержатся в элементах. Это может быть как примитивный тип языка C++, так и тип данных, определённый пользователем. В определении идентификаторов необходимо указать их атрибуты. В текущей реализации тип атрибутов может быть только примитивный целочисленный тип int, а их число ограничено пятью. Также в текстовом описании разрешены однострочные комментарии, начало которых обозначается двумя косыми чертами.

Определение класса графа генерируется с помощью транслятора из текстового представления алгоритма. Транслятор производит заголовочный C++ файл, который включается разработчиком в программу. Также транслятор генерирует текстовый файл с шаблонами определения шагов, которые могут сильно помочь в работе с интерфейсом Intel® Concurrent Collections.

Шаги описываются как обычные C++ функции, но имеют определённую сигнатуру, пример шага «Решение» приложения Blackscholes представлен на рисунке 5.

Возвращаемое значение – это одна из двух специальных констант: CNC_Success и CNC_Failure. Первое сигнализирует об успешном завершении вычислений, тогда как при возвращении CNC_Failure шаг не считается завершённым и его исполнение будет повторено позже.

Аргументами функции шага являются граф и идентификатор исполняемого шага. Граф предоставляет доступ к необходимым пространствам элементов данных с помощью методов Get и Put, первый служит для получения элемента данных по его идентификатору, второй – для добавления элемента данных с указанным идентификатором. В пространстве идентификаторов можно лишь добавлять новые экземпляры с помощью метода Put.

```

StepReturnValue_t Solve(
    blackscholes_graph_t& graph,
    const Tag_t& step_tag) {
    Data* parameters =
        graph.parameters.Get(step_tag);
    double result =
        SolveBlkSchlsEquation(parameters);
    graph.result.Put(step_tag, result);
    return CNC_Success;
}

```

Рисунок 5. Описание шага «Решение» в приложении Blackscholes

Для запуска алгоритма требуется создать объект графа данного алгоритма, инициализировать пространства элементов данных и идентификаторов входными данными и вызвать у графа метод run.

6. Перезапуск шагов

Порядок выполнения шагов неопределён, поэтому могут возникать ситуации, когда запущенный на выполнение шаг требует ещё не доступные данные. В этом случае продолжение выполнения шага невозможно, он прерывается и помещается в локальную очередь ожидания этого элемента данных. Это означает, что в момент, когда элемент данных станет доступен, все ожидавшие его шаги будут перезапущены.

Такой механизм позволяет значительно упростить работу программиста, но и накладывает дополнительные ограничения на определение шага:

- Все методы Get должны вызываться раньше любых методов Put
- До выполнения всех методов Get нельзя выделять или освобождать память
- Шаги не должны иметь сторонних эффектов

Не соблюдение этих ограничений может привести к утечке памяти и непредсказуемой работе программы.

Механизм перезапуска шагов позволяет запускать доступные шаги в произвольном порядке, не влияя на корректность программы.

7. Использование блочных алгоритмов

Несмотря на простоту в применении модели, часто можно получить значительный выигрыш в производительности, выполняя операции не для отдельных объектов, а для блоков объектов. Особенно это становится важно, когда вычисления для одного объекта малы, менее 5000 инструкций. В этом случае накладные расходы на создание и извлечение элементов данных и

идентификаторов из их пространств, создание и запуск шагов становятся очень весомыми.

В приложении Blackscholes решение дифференциального уравнения для одного набора входных параметров занимает менее 500 инструкций процессора. На рисунке 6 представлено сравнение времени выполнения следующих вариантов алгоритма Blackscholes для решения задачи на восьми потоках для 5000000 различных наборов параметров:

- Последовательная программа
- Один набор параметров за шаг
- 10 наборов параметров за шаг
- 100 наборов параметров за шаг
- 1000 наборов параметров за шаг
- Явное использование системных потоков

При явном использовании программистом системных потоков путем системных вызовов ОС для распараллеливания Blackscholes, разделении объёма работ поровну между всеми потоками, может быть достигнута производительность на 2% лучше, чем при использовании Intel® Concurrent Collections.

На рисунке 7 представлены графики производительности разных вариантов алгоритма Blackscholes по отношению к версии, использующей системные потоки, исполняющейся на одном потоке. Масштабируемость версии алгоритма с размером блоков 200 меньше на 2%, что обусловлено накладными расходами на синхронизацию при доступе к структурам, реализующим пространства элементов данных и идентификаторов.

На рисунке 8 представлены графики времени выполнения программы Dedup из набора приложений для тестирования

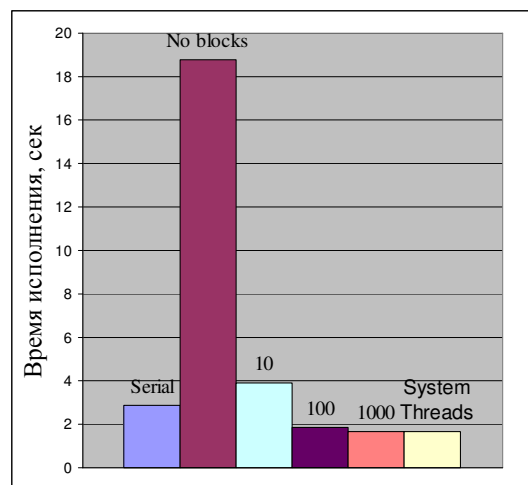


Рисунок 6. Сравнение разных вариантов алгоритма Blackscholes

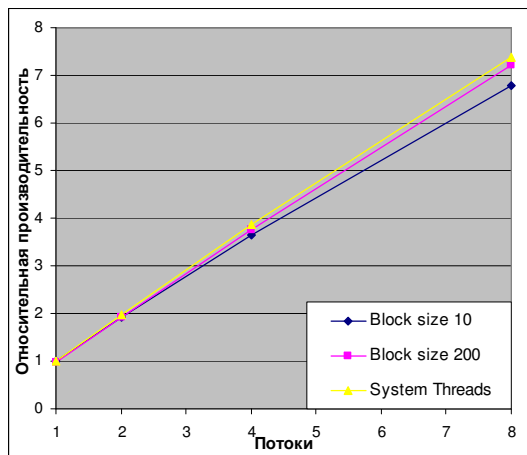


Рисунок 7. Масштабируемость разных вариантов алгоритма Blackscholes

производительности системы PARSEC и этой же программы в представлении Intel® Concurrent Collections. Dedup производит сжатие файлов методом deduplication. Алгоритм сжатия достаточно сложен, представляет собой конвейер, в котором элементы последней стадии, записи в файл, должны обрабатываться строго в последовательном порядке. Масштабируемость алгоритма в представлении Intel® Concurrent Collections невелика из-за большого числа необходимых зависимостей в графе.

Ведутся разработки по улучшению планировщика заданий путём назначения шагам приоритетов. Эта возможность позволяет уменьшить время исполнения программы Dedup

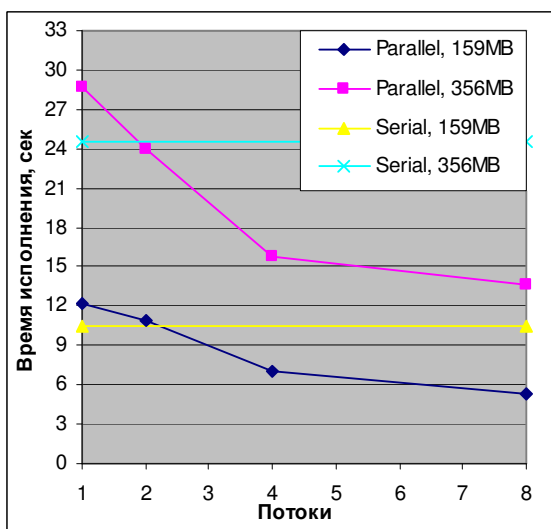


Рисунок 8. Время исполнения Dedup, последовательной и параллельной версии, на файлах размера 159MB и 356MB

на восьми в потоках в 1,4 раза.

8. Заключение

Была представлена новая модель параллельных вычислений, имеющая следующие цели:

- Поддержка всех типов параллелизма
- Поддержка всего спектра параллельных архитектур
- Поддержка всего спектра моделей исполнения
- Соккрытие низкоуровневых вопросов параллельного программирования

Программа в этой модели состоит из абстрактного представления параллельного алгоритма и высокоуровневых операций и структур данных, реализованных на последовательном языке программирования.

Основными преимуществами использования описанного подхода к построению параллельных программ являются:

- Приложение легко портировать, потому что они не являются платформенно-зависимыми
- Более простая и быстрая разработка, что обеспечивается сокращением вопросов параллелизма и выделением высокоуровневого описания алгоритма
- Вопросы производительности на конкретной платформе занимается реализация Intel® Concurrent Collections

Применение модели к стандартным тестам производительности показывает отличную масштабируемость в случаях, если вычисления не разбиваются на слишком мелкие шаги и зависимости по данным не образуют сложных графов с узкими местами в виде набора последовательных вычислений. Для более сложных случаев исследуется возможность автоматического объединения нескольких шагов и создания более эффективного планировщика вычислений.

9. Литература

[1] R. D. Blumofe and C. F. Joerg, "Cilk: An Efficient Multithreaded Runtime System", Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 207-216. <http://supertech.csail.mit.edu/papers/cilkjpd96.pdf>.

[2] "OpenMP Application Program Interface, Version 2.5 May 2005", from the OpenMP web site: <http://www.openmp.org>.

[3] J. Reinders, "Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism", 2007.

[4] C. Bienia, S. Kumar, J. P. Singh and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications", Princeton University, TR-811-08.