DOI 10.37882/2223–2966.2020.05.43

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ВЫЧИСЛИТЕЛЬНОГО ГРАФА ДЛЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

THE IMPLEMENTATION OF THE COMPUTATIONAL GRAPH FOR PARALLEL COMPUTING

D. Chaikovsky
N. Gulevich
E. Pchelinceva

Summary. The subject of research is the analysis of software implementations of a computational graph in C++ using the Cpp libraries-Taskflow, Task dependency of the OpenMP standard and Intel Threading Building Blocks (TBB). The features of the software implementation of the computational graph using the open source library Cpp-Taskflow, which is written for C++, are considered. The relevance of the research is determined by the wide spread of big data processing technologies, parallel programming and the need to study and create appropriate tools, including for the software implementation of computational graphs. The paper identifies the main limitations faced by developers of parallel programs. Conclusions are made that the implementation of a computational graph in Cpp-Taskflow has advantages over existing tools.

Keywords: parallel computing, parallel programming, computational graph, dataflow architecture, app-Taskflow library, parallel programming on Cpp, object-oriented programming, OpenMP, Intel TBB, methods describing parallelism.

Чайковский Дмитрий Станиславович
К.ф.-м.н., доцент, Саратовская государственная юридическая академия
chaikovskyds@gmail.com
Гулевич Наталья Анатольевна
К.т.н., доцент, Саратовский государственный технический университет
gulevich005@mail.ru
Пчелинцева Елена Германовна
К.с.н., доцент, Саратовский государственный технический университет
alenapchelka@gmail.com

Аннотация. Предметом исследования работы является анализ программных реализаций вычислительного графа в среде C++ средствами библиотек Cpp-Taskflow, Task dependency стандарта OpenMP и Intel Threading Building Blocks (TBB). Рассмотрены особенности программной реализации вычислительного графа средствами библиотеки с открытым исходным кодом Cpp-Taskflow, которая написана для C++. Актуальность исследования определяется широким распространением технологий обработки больших данных, параллельного программирования и необходимостью изучения и создания соответствующих средств, в том числе для программной реализации вычислительных графов. В работе выявлены основные ограничения, с которыми сталкиваются разработчики параллельных программ. Сформулированы выводы о том, что реализация вычислительного графа в Cpp-Taskflow обладает преимуществами перед существующими средствами.

Ключевые слова: параллельные вычисления, параллельное программирование, вычислительный граф, dataflow-архитектура, библиотека Cpp-Taskflow, программирование в Cpp, объектно-ориентированное программирование, OpenMP, Intel TBB, методы описания параллелизма.

Введение

В настоящее время, параллельные вычисления из узконаправленной дисциплины трансформировались в обязательные знания для разработчика современного программного обеспечения.

Совершенствование методов программного описания параллелизма сложных процессов и систем является основной задачей в параллельном программировании [1].

Колоссальные объемы данных, которые генерируют всевозможные устройства и средства их обработки получили название «большие данные». Технологии обработки больших данных неразрывно связаны с параллельными вычислениями. Такие мощные методы обработки больших данных, как Google.MapReduce и Apache Hadoop базируются на параллельных вычислениях [2].

В параллельном программировании, успешно развивается система, основанная на передаче сообщений. Она является одним из способов организации взаимодействия элементов в параллельных вычислениях и в объектно-ориентированном программировании. В роли сообщений могут выступать данные или управляющие сигналы. Такую систему можно представить в виде узлов, которые обрабатывают сообщения и набора связей между ними. Такая модель получила название вычислительный граф [3, 4].

Вычислительный граф обеспечивает взаимосвязь между задачами и позволяет реализовать dataflow-архитектуру [5].

Архитектура dataflow представляет собой управление потоком данных, в котором отсутствуют последовательные инструкции. Программа, написанная в такой системе, представляет собой вычислительный граф, а не набор команд. В системах dataflow передача и хранение данных осуществляется с помощью токенов, которые можно рассматривать как структуру, состоящую из передаваемого значения и указателя узла назначения (метки).

К достоинству dataflow-архитектуры можно отнести ее масштабируемость. Устройства объединяются коммутаторами, а узлы равномерно распределяются между устройствами. Архитектура с управлением потоком данных может быть статической и динамической. В статической модели потоковых вычислений каждый узел существует в единственном экземпляре, а количество узлов и токенов заранее определено. Динамическая модель потоковых вычислений может содержать узлы, которые имеют некое количество экземпляров. Токен содержит дополнительный параметр, необходимый для идентификации, при адресации в разные экземпляры одного узла. В динамической архитектуре становится возможным создание рекурсий, процедур и распараллеливание циклов. Распараллеливание циклов возможно, если отсутствует зависимость от данных.

В настоящее время широкое распространение получили программные реализации вычислительного графа в среде C++ с помощью функций библиотек OpenMP — Task dependency и Intel Threading Building Blocks (TBB) — Flow Graph [6, 7]. Однако, есть множество ограничений в их использовании. Например, OpenMP использует статические описания задач с последовательным выполнением, что усложняет обработку динамических потоков, где структура графа неизвестна во время программирования. В TBB описание графа задач реализуется достаточно сложно и часто приводит к большому количеству строк кода, чтение и отладка которых достаточно трудоемки.

Построение вычислительного графа в Cpp-Taskflow

Рассмотрим возможности библиотеки с открытым исходным кодом Cpp- Taskflow для среды C++, которая позволяет писать параллельные программы с использованием графов зависимостей [8].

Библиотека Cpp-Taskflow использует возможности языка C++ 17, это позволяет задействовать новые функции C++ при написании параллельных программ. Язык описания графов в Cpp-Taskflow позволяет создавать статические и динамические графовые конструкции при написании параллельных алгоритмов. Приведем пример программы с использованием Cpp-Taskflow, в которой, создадим граф зависимостей из четырех задач, A, B, C, D. Введем ограничения: задача A выполняется до задачи B и задачи C, а задача D выполняется после задачи B и задачи C.

В начале создается объект для генерации задач: «tf:: Taskflow tf;». Объект «taskflow» позволяет создавать графы зависимостей задач и отправлять их в потоки для выполнения.

Затем генерируется задача A:

«tf:: Task A = tf.emplace([](){std:: cout << «Task A» << std:: endl;});»

Создание и обработка задачи могут быть написаны отдельно:

«tf:: Task A = tf.emplace([](){}); A. work([] () {std:: cout << "Task A" << std:: endl;});»

Метод «emplace» создает задачу из заданного вызываемого объекта. Причем, можно создавать несколько задач одновременно:

«tf:: Task A = tf.emplace([](){std:: cout << «Task A» << std:: endl;});
tf:: Task B = tf.emplace([](){std:: cout << "Task B" << std:: endl;});
tf:: Task C = tf.emplace([](){std:: cout << "Task C" << std:: endl;});
tf:: Task D = tf.emplace([](){std:: cout << "Task D" << std:: endl;});»

Задачи также можно создавать в совокупности, следующим образом:

«auto [A, B, C, D] = tf.emplace(
[] () {std:: cout << "Task A" << std:: endl;},
[] () {std:: cout << "Task B" << std:: endl;},
[] () {std:: cout << "Task C" << std:: endl;},
[] () {std:: cout << "Task D" << std:: endl;});»

Каждый раз, когда создается задача, объект «taskflow» добавляет узел к существующему графу и возвращает дескриптор задачи. Дескриптором задачи являются объекты класса, которые заключают определенный узел. Каждый узел имеет универсальную оболочку для хранения и вызова любой задачи.

После того, как задачи будут созданы, следующим шагом является добавление зависимостей. Зависи-

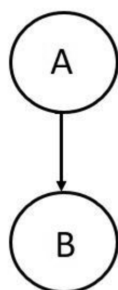

Рис. 1. Порядок обработки задач А и В

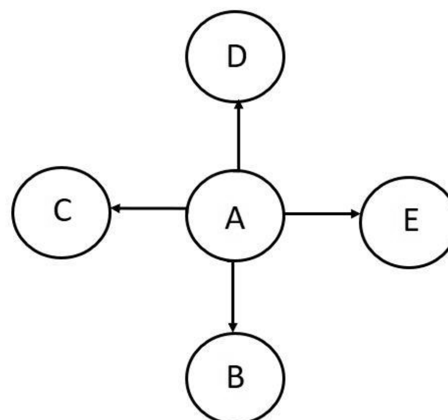

Рис. 2. Задачи B, C, D, E обрабатываются параллельно после задачи А

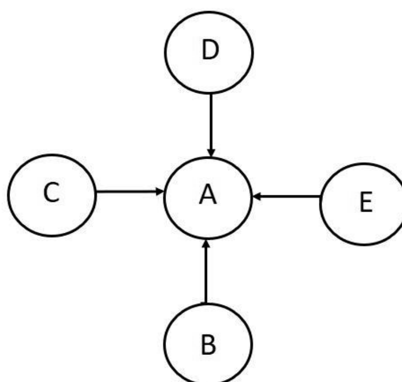

Рис. 3. Задачи B, C, D, E обрабатываются параллельно перед задачей А

мость задачи — это направленное ребро от задачи А к задаче В таким образом, что задача А выполняется перед задачей В. Иными словами, узел В не будет вызывать свою задачу, пока узел А не завершит свою задачу.

Рассмотрим различные формы записи обработки задач.

Существуют две формы записи реализации, изображенной на рис. 1: задача А запускается перед задачей В: «A.precede(B);»; задача В запускается после задачи А:

«B.gather(A);».

Данная форма записи применима и ко множеству задач (рис. 2).

Схема, изображенная на рис. 2 записывается следующим образом:

«A.precede(B, C, D, E);»

Аналогичным образом можно записать решение задачи А после параллельной обработки задач B, C, D и E (рис. 3).

Схема, изображенная на рис. 3 записывается: «A.gather(B, C, D, E);».

Отладка параллельной программы, написанной с помощью Cpp-Taskflow достаточно проста. Программист может визуализировать поток каждой задачи. Покажем это на примере.

Например, есть следующая задача:

```
«#include «./taskflow/taskflow.hpp»
int main(){
tf:: Taskflow tf; tf:: Task A = tf.emplace([] () {}).name(“A”);
tf:: Task B = tf.emplace([] () {}).name(“B”);
```

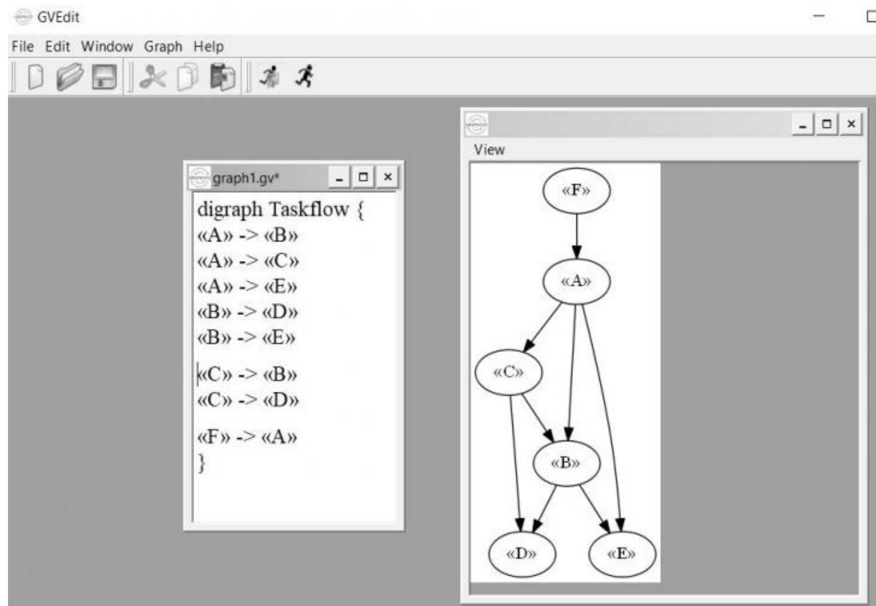

Рис. 4. Визуализация графов с помощью программы GraphViz

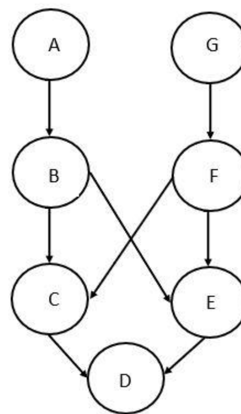

Рис. 5. Граф, состоящий из семи задач и восьми зависимостей

```
tf:: Task C = tf.emplace([] () {}).name("C");
    tf:: Task D = tf.emplace([] () {}).name("D");
    tf:: Task E = tf.emplace([] () {}).name("E");
    tf:: Task F = tf.emplace([] () {}).name("F");
    A.precede(B, C, E);
    C.precede(B, D);
    B.precede(D, E);
    F.precede(A);
    tf.dump(std:: cout);
    return 0;}».
```

Дамп текущего графа задач генерируется с помощью команды «tf.dump(std:: cout);» и выглядит следующим образом:

«digraph Taskflow {«A» -> «B» «A» -> «C» «A» -> «E» «B» -> «D» «B» -> «E» «C» -> «B» «C» -> «D» «F» -> «A»}».

Для того, чтобы нарисовать диаграммы из сгенерированного кода можно использовать программу GraphViz [9] или аналогичные [10], которые представляют собой набор средств по автоматической визуализации графов (рис. 4).

Вышеописанный пример демонстрирует возможность визуализации потока задач, что позволяет программисту находить ошибки на ранней стадии и ускорять процесс отладки кода.

Библиотека Cpp-Taskflow позволяет реализовать различные методы распараллеливания: параллельные циклы, алгоритмы графов и динамические потоки.

Библиотека имеет унифицированный интерфейс как для статических, так и для динамических задач. В Cpp-

Таблица 1. Сравнение программных реализаций графа, изображенного на рис. 5.

Cpp-Taskflow	OpenMP	TBB

tf:: Taskflow tf;
auto [g, f, e, d, a, b, c] = tf.emplace(
[] () {std:: cout << ″g\n ″;},
[] () {std:: cout << ″f\n ″;},
[] () {std:: cout << ″e\n ″;},
[] () {std:: cout << ″d\n ″;},
[] () {std:: cout << ″a\n ″;},
[] () {std:: cout << ″b\n ″;},
[] () {std:: cout << ″c\n ″;},);
g. precede(f);
f. precede(e, c);
e. precede (d);
a. precede(b);
b. precede(e, c);
c. precede(d);
tf. wait_for_all();
	#pragma omp parallel
	{
	#pragma omp single
	{
	int g_f, f_e, f_c, e_d;
	int a_b, b_c, b_e, c_d;
	#pragma omp task depend (out g_f)
	std:: cout << ″g\n ″;
	#pragma omp task depend (out: a_b)
	std:: cout << ″a\n ″;
	#pragma omp task depend (in : g_f)
	depend (out:
	f_e, f_c)
	std:: cout << ″f\n ″;
	#pragma omp task depend (in: a_b) depend
	(out:
	b_c, b_e)
	std:: cout << ″b\n ″;
	#pragma omp task depend (in: f_e, b_e)
	depend (
	out: e_d)
	std:: cout << ″e\n ″;
	#pragma omp task depend (in: f_c, b_c)
	depend (
	out: c_d)
	std:: cout << ″c\n ″;
	#pragma omp task depend (in: e_d, c_d)
	std:: cout << ″d\n ″;
	}}
		using namespace tbb;
		using namespace tbb:: f low;
		int n = task_ scheduler_ init:: default_num
		_threads();
		task_ scheduler_ init init(n);
		graph g;
		continue_node<continue_msg> g (g, [] (const
		continue_msg &) {
		std:: cout << ″g\n ″;});
		continue_node<continue msg> f (g, [] (const
		continue_msg &) {
		std:: cout << ″f\n ″;
		});
		continue_node<continue msg> e (g, [] (const
		continue_msg &) {
		std:: cout << ″e\n ″;
		});
		continue_node<continue msg> d (g, [] (const
		continue_msg &) {
		std:: cout << ″d\n ″;
		});
		continue_node<continue msg> a (g, [] (const
		continue_msg &) {
		std:: cout << ″a\n ″;
		});
		continue_node<continue msg> b (g, [] (const
		continue_msg &) {
		std:: cout << ″b\n ″;
		});
		continue_node<cont inue msg> c (g, [] (const
		continue_msg &) {
		std:: cout << ″c\n ″;
		});
		make_edge (g, f);
		make_edge (f, e);
		make_edge (f, c);
		make_edge (e, d);
		make_edge (a, b);
		make_edge (b, c);
		make_edge (b, e);
		make_edge (c, d);
		g. try_put (continue_msg ());
		a. try_put (continue_msg ());
		g. wait_ for_ all();

Taskflow используется объектно-ориентированный подход: задача в Cpp-Taskflow определяется как вызываемый объект, для которого выполняется операция «std:: invoke».

Сравнение программных реализаций потоков данных

Для сравнения Cpp-Taskflow с OpenMP и TBB опишем граф, изображенный на рисунке 5 в каждой из них.

На рис. 5 показан пример графа, а в таблице 1 показана его реализация с помощью Cpp-Taskflow, OpenMP и TBB соответственно.

Сравнивая листинги в таблице 1, можно сделать вывод, что реализация графа, представленного на рис. 5 наиболее кратко записана при помощи Cpp-Taskflow.

При использовании OpenMP необходимо указывать условие зависимости на обеих сторонах узла графа

и определить правильный топологический порядок для описания каждой задачи таким образом, чтобы он соответствовал последовательному потоку.

Реализация графа с помощью TBB достаточно объемна. Необходимо использовать шаблоны классов «continue_node» и «message» даже при решении простых задач. Чтобы запустить вычислительный граф, необходимо явно задать TBB исходные задачи и вызвать метод «try_put», чтобы активировать ввод данных. Все это увеличивает трудоемкость программирования.

В Cpp-Taskflow каждый объект «taskflow» содержит один граф зависимостей задачи. После того, как граф зависимостей задачи решен, следующим шагом является отправка его в потоки для выполнения. Граф существует и остается под контролем до тех пор, пока пользователи не отправят его на выполнение. Отправленный на выполнение граф Cpp-Taskflow представляет собой структуру данных, которая хранит метаданные, полученные во время выполнения. Каждый объект «taskflow» имеет структуру данных для отслеживания состояния выполнения отправленных графов. Связь основана на паре c++ «shared_future» и «promise». Программисты могут получить эту информацию для проверки и отладки графа. Все задачи выполняются в общем хранилище потоков, связанном с исполнителем, чтобы отследить, какой поток какую задачу выполняет.

В Cpp-Taskflow есть два способа отправки графа зависимостей: блокирующие и неблокирующие выполнение. В первом необходимо использовать команду «wait_for_all». Он отправляет граф потокам и блокам до тех пор, пока все задачи не завершатся. Второй способ отправляет граф потокам и сразу же возвращается в программу, не дожидаясь завершения всех задач. Это позволяет программистам выполнять другие вычисления.

Средства Cpp-Taskflow можно также использовать в динамическом программировании. Динамическое программирование сводится к созданию графа зависимостей во время выполнения задачи. Динамические задачи создаются на основе запущенного графа. Эти задачи порождаются из родительской и группируются вместе, чтобы сформировать граф зависимостей задач, который можно называть подпотоком. Преимущество библиотеки Cpp-Taskflow, при решении задач подобного типа в том, что она предоставляет унифицированный интерфейс для статических и динамических задач.

Cpp-Taskflow использует контейнер типов «std:: variant» как для статических, так и для динамических графовых конструкций. Динамику зависимости можно описать используя метод «emplace», с дополнительным аргументом «tf:: SubflowBuilder». По умолчанию, порожденный подпоток присоединяется к своей родительской задаче. Это заставляет подпоток следовать последующим ограничениям зависимостей своей родительской задачи. Можно отсоединить подпоток от его родителя с помощью метода detach, в этом случае его выполнение будет проходить независимо. Обособленный подпоток в итоге присоединится к концу топологии своей родительской задачи.

В Cpp-Taskflow можно создать планировщик для решения конкретных задач. Совместное использование исполнителя среди нескольких объектов «taskflow» облегчает модульные разработки в больших приложениях, избегая при этом проблем, связанных с избыточной нагрузкой на поток. Для управления исполнителем используется объект «std:: shared_ptr».

Для быстрого создания объемных параллельных программ из более простых шаблонов, в Cpp-Taskflow предусмотрена инкапсуляция алгоритмов в шаблоны. Cpp-Taskflow имеет встроенные наборы алгоритмов, которые реализуют общие параллельные рабочие нагрузки, такие как «parallel_for», «reduce» и «transform». Программисты могут легко писать универсальный код с помощью шаблона и объединять его с графами зависимостей своих задач для создания более крупных прикладных модулей.

Заключение

В результате исследования сопоставлены основные концепции реализации параллельных вычислений с помощью вычислительного графа в среде C++ (OpenMP и TBB) с новой библиотекой Cpp-Taskflow. Выявлены некоторые особенности OpenMP и TBB, которые увеличивают трудоемкость написания параллельных программ. Например, обработать динамические потоки в OpenMP, из-за статического описания задач с последовательным выполнением будет достаточно сложно. В Intel TBB запись задачи сопровождается объемным описанием, что приводит к большому количеству строк кода, в результате отладка программ становится трудоемкой.

Библиотека Cpp-Taskflow позволяет создавать как статическую, так и динамическую структуру графа, причем интерфейс является унифицированным. Cpp-Taskflow предоставляет возможность инкапсуляции алгоритмов в шаблоны. Для визуализации потока задач имеется возможность генерации дампа текущего графа задач, что позволит ускорить процесс отладки кода.

В работе приведены программные коды, реализующие вычислительный граф с помощью Cpp-Taskflow, OpenMP и TBB. Несмотря на то, что анализ эффективности программ является сложной задачей, которая учи-

тывает множество факторов, можно сделать вывод, что программа с библиотекой Cpp-Taskflow отличается компактностью и простотой описания потока задач.

К настоящему времени создан целый ряд проектов, в которых используется Cpp-Taskflow: анализатор времени выполнения для технологии создания сверхбольших интегральных схем, система распределенного программирования на параллельных потоках данных, инструмент для параллельного программирования на GPU [11]. Это подтверждает целесообразность использования возможностей библиотеки при решении сложных задач, связанных с параллельными вычислениями.

ЛИТЕРАТУРА

1. Котов В. Е. Проблемы развития параллельного программирования // Проблемы информатики. — 2016. — № 2. — С. 70–78.
2. Чайковский Д. С. Средства обработки больших данных // Современная наука: актуальные проблемы теории и практики. Серия: Естественные и технические науки. — 2018. — № 12. — С. 101–105.
3. Воеводин, В. В. Параллельные вычисления / В. В. Воеводин, Вл. В. Воеводин. — СПб.: БХВ-Петербург, 2002. — 608 с.
4. Руденко Ю. М. Представление операторов выбора и цикла языков программирования в граф-схемах алгоритмов // Инженерный журнал: наука и инновации. — 2013. — № 11. — С. 1–7. — DOI: 10.18698/2308–6033–2013–11–1066
5. Arthur H. Veen Dataflow Machine Architecture // ACM Computing Surveys (CSUR). — 1986. — vol. 18, issue 4. — P. 365–396. — DOI: 10.1145/27633.28055
6. OpenMP. URL: http://openmp.org (дата обращения: 07.04.2020).
7. Intel Threading Building Blocks. URL: https://software.intel.com/en-us/tbb (дата обращения: 07.04.2020).
8. Cpp-Taskflow. URL: https://github.com/cpp-taskflow (Дата обращения: 13.03.2020).
9. Graphviz-Graph Visualization Software. URL: https://www.graphviz.org (дата обращения: 07.04.2020).
10. Viz-js. URL: http://viz-js.com (дата обращения: 07.04.2020).
11. Who is Using Cpp-Taskflow. URL: https://github.com/cpp-taskflow/cpp-taskflow#who-is-using-cpp-taskflow (дата обращения: 07.04.2020).

© Чайковский Дмитрий Станиславович (chaikovskyds@gmail.com),
Гулевич Наталья Анатольевна (gulevich005@mail.ru), Пчелинцева Елена Германовна (alenapchelka@gmail.com).
Журнал «Современная наука: актуальные проблемы теории и практики»

Саратовская государственная юридическая академия