электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.75+519.171.2

Оценка эффективности выполнения параллельных алгоритмов операций на графах в асинхронном представлении

Петров Ю.К., *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Компьютерные системы и сети»

Научный руководитель: Иванова Г.С., д.т.н., профессор,
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
bauman@bmstu.ru

Введение

Параллельные алгоритмы являются быстро развивающимся направлением теории алгоритмов. Разработка параллельных алгоритмов начинается в конце 70-ых годов XX века. Однако задача эффективного распараллеливания алгоритмов операций над графами считается одной из нерешённых до конца по следующим причинам:

- возможная гетерогенность вычислительных систем – изначально вычислительные системы для научных целей собирались на заказ из одинаковых компонентов, с ростом тенденций распределённых вычислений и метакомпьютинга, естественным является предположение о гетерогенности вычислительных систем, однако существующие решения, как правило, предполагают как минимум единую архитектуру процессоров параллельной вычислительной системы;
- проблема балансировки нагрузки – граф разбивается на домены статически, что приводит к неравномерности распределения нагрузки по вычислительным узлам: нагрузка распределена равномерно лишь в том случае, когда число обращений к вершинам одинаково.

Одной из проблем реализации высокоэффективных параллельных алгоритмов являлось отсутствие модели представления графов, изначально рассчитанной на параллельные системы. Основные концепции современных моделей сформировались примерно в середине 80-ых годов, что не позволяет им использовать многие преимущества современных параллельных систем. Предлагаемая модель (называемая

далее асинхронной) изначально рассчитана на параллельные (в том числе и распределённые) системы.

Асинхронная модель представления графов

Основными концепциями предложенной модели являются:

- асинхронность – нет никаких гарантий выполнения операции в точно определенное время, однако существует гарантия отсутствия блокировки управляющих процессов;
- горизонтальная масштабируемость – возможно добавление практически неограниченного числа узлов без потери эффективности;
- виртуализация – все процессы являются легковесными процессами виртуальных машин, что снимает проблему гетерогенности систем;
- делегирование обязанностей.

Простейшая модель графа в данном представлении показана на рисунке 1.

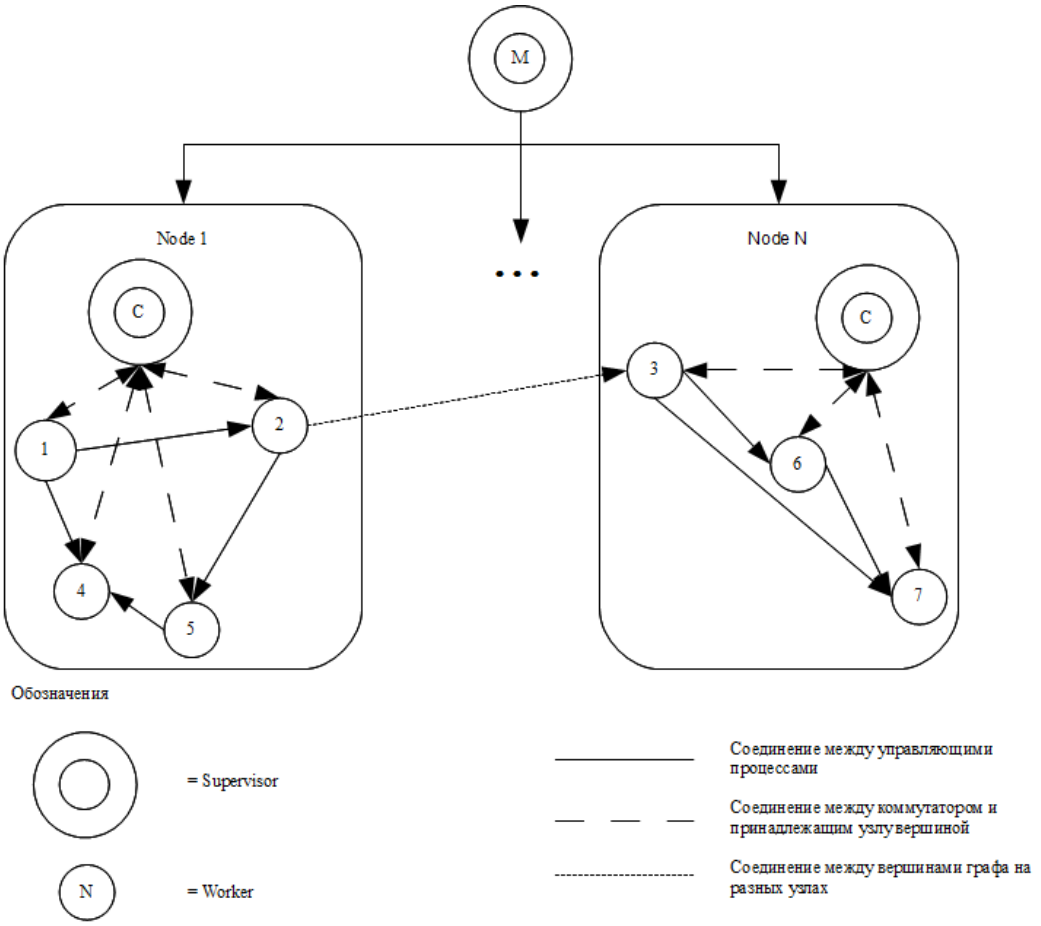


Рис.1. Распределённый по двум узлам граф в простейшем асинхронном представлении

Под наблюдателями (supervisor) подразумеваются процессы, способные к перехвату сообщений состояния дочерних процессов, например, перехват сообщений завершения процесса. В этом случае процесс-наблюдатель может сам предпринять действия по устранению данной ошибки, например, перезапустить дочерний процесс. Таким образом обеспечивается локализация ошибки и программная отказоустойчивость. Узлами (Nodes) являются виртуальные машины вне зависимости от того, где они расположены и как соединены.

Модель является иерархической, иерархия процессов модели представлена на рисунке 2.

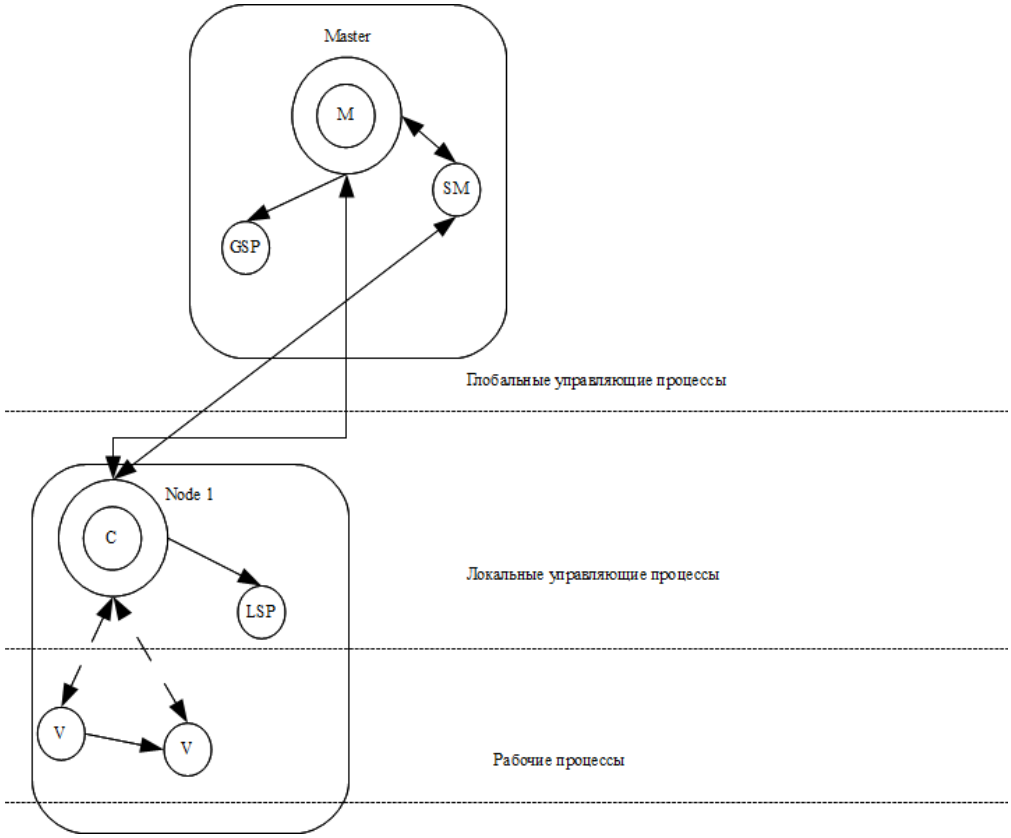


Рис.2. Иерархия процессов асинхронной модели

Иерархия содержит следующие виды процессов:

- мастер Master (M) – существует в единственном экземпляре, порождает все другие процессы, отвечает за создание узлов;
- поддерживающий мастер Support Master (SM) – необязательный тип процессов, используется для децентрализации управления и обеспечивает более высокую степень

утилизации ресурсов в случае временной блокировки мастера (хотя модель и является асинхронной, возможно построение синхронных операций на базе асинхронных);

- глобальный вспомогательный процесс Global Service Process (GSP) – вспомогательный процесс уровня глобальных управляющих процессов, не имеет доступа к локальным данным узлов, является необязательным;
- коммуникатор Communicator (C) – является главным процессом узла, отвечает за размещение вершин и их отказоустойчивость;
- локальный вспомогательный процесс Local Service Process (LSP) – вспомогательный процесс уровня локальных управляющих процессов, имеет доступ только к локальным данным узлов, является необязательным;
- вершина Vertex (V) – рабочий процесс, представляет собой вершину графа, способен к обмену сообщениями с другими процессами в модели.

Вспомогательные процессы

Одной из ключевых особенностей модели являются вспомогательные процессы. Данный тип процессов может использоваться для любых целей, например:

- ведение журнала операций над вершинами узла;
- ведение журнала об узлах: подключение, аварийный перезапуск;
- создание новых интерфейсов для работы с моделью, например, через интернет.

Следует заметить, что практически все операции в данном алгоритме происходят асинхронно. Таким образом, мастер может отправить новое задание сразу после отправки первого.

Вспомогательные процессы могут быть как рабочими процессами, так и наблюдателями, в случае, если вспомогательный процесс порождает собственные вспомогательные процессы. Однако в этом случае вспомогательный процесс не обязан быть наблюдателем: за локализацию ошибки будет отвечать первый наблюдатель, которым будет являться коммуникатор.

Главным требованием к вспомогательным процессам является соблюдение основных принципов модели. Вспомогательный процесс не должен вносить дополнительную синхронность, если этого явно не требует алгоритм. Вспомогательный процесс может быть аварийно завершён, в этом случае за отказоустойчивость системы и корректность дальнейших операций должен отвечать породивший вспомогательный процесс коммуникатор (или мастер). Вспомогательные процессы должны создаваться только через обработчик сообщений.

Практическая часть

Для опытов был использован следующий алгоритм:

1. Мастер получает запрос на поиск максимального веса ребра.
2. Мастер создаёт глобальный вспомогательный процесс, который получает в качестве аргументов два параметра: количество узлов и pid мастера.
3. Мастер отправляет этот запрос коммуникаторам.
4. Коммуникатор, получивший запрос, создаёт локальный вспомогательный процесс, который получает в качестве аргументов pid коммуникатора и число вершин узла.
5. Локальный вспомогательный процесс создаёт у себя две ETS-таблицы типа «множество»: одну для пройденных вершин, другую для весов рёбер.
6. Коммуникатор передаёт одной из вершин сообщение, содержащее запрос и pid локального вспомогательного процесса.
7. Вершина, получившая запрос, отправляет локальному вспомогательному процессу веса всех своих рёбер и свой ID, при этом рассылая запрос всем соседним вершинам, после чего временно игнорирует запрос на максимальный вес ребра.
8. Локальный вспомогательный процесс, получив сообщение, добавляет вес ребра в таблицу весов, ID вершины – в таблицу пройденных вершин. Если размер таблицы пройденных вершин равен количеству вершин на узле – локальный вспомогательный процесс находит максимальный вес в таблице и отправляет его коммуникатору. Если нет – происходит новая итерация.
9. Коммуникатор, получивший сообщение с результатом, отправляет это сообщение глобальному вспомогательному процессу.
10. Глобальный вспомогательный процесс, получив сообщение, производит действия, аналогичные тем, что производил локальный вспомогательный процесс в пункте 8.
11. Мастер возвращает результат (возврат результата также возможен глобальным вспомогательным процессом напрямую).

Эксперименты проводились на SMP-системе (процессор Intel Core2Quad, 2,66 ГГц), Erlang R15B02, настройки планировщика стандартные. Проводилось по 100 экспериментов для каждого числа вершин. Время измерялось с помощью функции `timer:tc/3`, следует заметить, что в некоторых случаях она может давать погрешность до 13-14 мс. Графы были размещены в DETS-таблицах, однако время размещения графа

не вошло в тесты. Степень вершины графа – 2,6. На рисунке 3 показана зависимость размера DETS-таблицы от количества вершин, вид записи – {ID, ListOFConnections, ListOfNeighbors}, где ListOfConnections/ListOfNeighbors – списки, элементами ListOf Connections являются кортежи вида {ID, Weight}, где Weight – вес ребра между данной вершиной и вершиной, для которой задан этот параметр, ListOfNeighbors – список ID вершин, указывающих на данную вершину.

Рис.3. Зависимость размера DETS-таблицы от количества вершин.

Для каждого алгоритма проводилось по 100 экспериментов, доверительный интервал составляет 196 мкс с вероятностью 95%.

Зависимость времени выполнения операции от количества вершин графа показана на рисунке 4. Резкое увеличение времени работы при 1000 вершин объясняется настройками планировщика виртуальной машины Erlang, которые по умолчанию не позволяют поддерживать столько процессов в активном состоянии, отправляя их в состояние «сна».

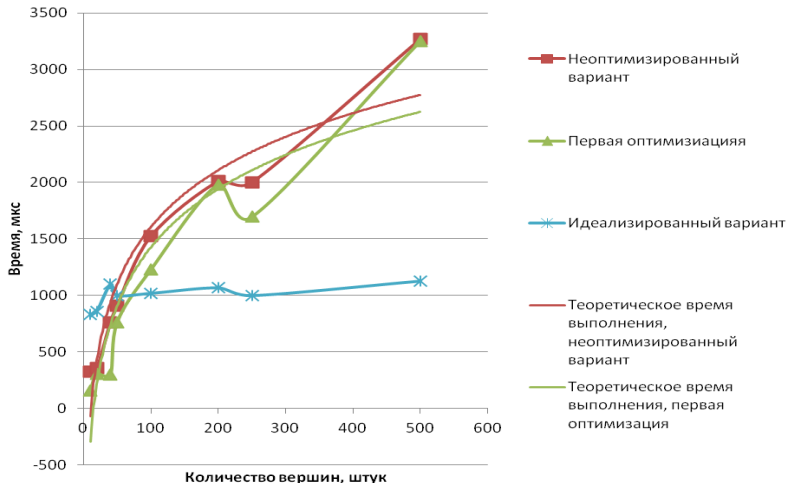
Теоретически, за счёт увеличения количества одновременно активных вершин, время выполнения алгоритма может быть уменьшено: в реализованном алгоритме коммуникатор отправлял сообщение только одной вершине, если же отправлять сообщение двум вершинам, расположенным достаточно далеко друг от друга, алгоритм должен работать быстрее. Результаты экспериментов с оптимизированной версией алгоритма также показаны на рисунке 4.

Для 1000 вершин оптимизированный алгоритм показал время работы на 33% больше, чем неоптимизированный. Это объясняется тем, что в случае оптимизированного алгоритма ещё больше вершин активны, планировщик ещё чаще отправляет вершины в

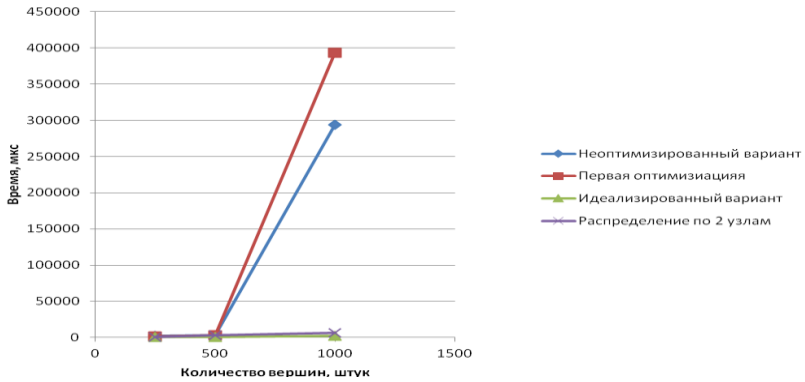
состояние «сна» и переводит их в активное состояние при поступлении сообщения. Снижение производительности для 10, 40 и 200 вершин объясняются погрешностями таймера.

Дополнительно были проведены тесты для идеализированного варианта, не требующего пересылки сообщений, так как все вершины находились в ОЗУ до начала алгоритма:

1. Из таблицы вершин выбирались все веса рёбер в отдельную ETS-таблицу типа «множество».
2. Производился поиск максимального значения.



а



б

Рис. 4. Сводный график времени работы различных алгоритмов на интервалах 10-500 вершин (а) и 250-1000 вершин (б)

Идеализированный вариант теоретически задействует также все 4 ядра, т.к. Erlang поддерживает SMP-системы и многие из его стандартных функций способны к автоматическому масштабированию. Этот вариант является теоретически идеальным, т. к. нет никаких дополнительных расходов.

Увеличение времени работы для 10-50 вершин объясняется высокими накладными расходами на синхронизацию потоков при работе с разделяемым ресурсом в ОЗУ. Как видно, 1000 вершин обрабатываются менее чем за 1% времени, требуемого на обработку того же количества вершин предыдущими алгоритмами.

Для 500 и 1000 вершин были проведены дополнительные испытания: граф распределялся по двум узлам, расположенным на одной SMP-системе. Результаты представлены в таблице. Графическая интерпретация результатов представлена на рисунке 5, для изображения тенденции изменения кривизны графиков добавлены результаты для 250 вершин.

Кол-во вершин	500	1000
Среднее время, мкс	3290	6730
В среднем на вершину, мкс	6,6	6,7
Ускорение относительно результата на одном узле, %	-5,4	977,1

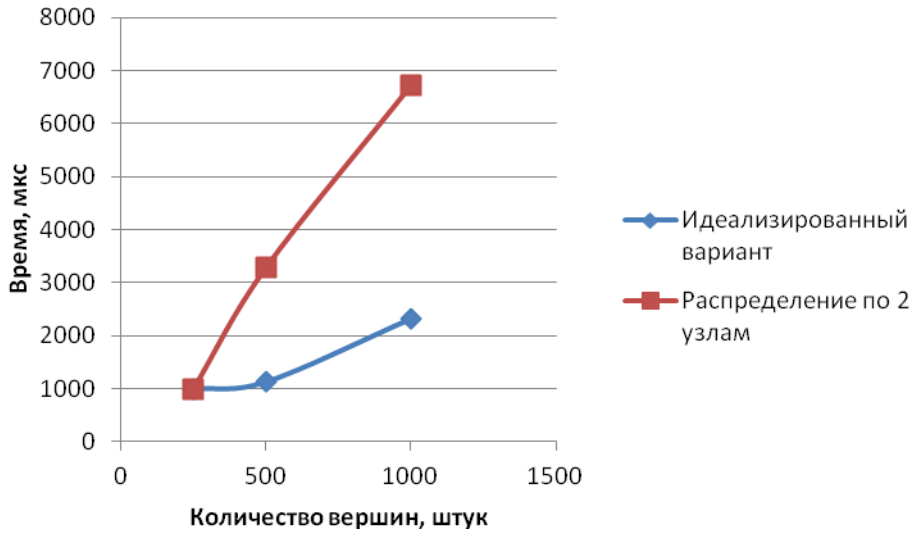


Рис. 5. Сравнение времени работы идеализированного и распределенного по двум узлам алгоритмов

Увеличение времени алгоритма при работе с 500 вершинами можно объяснить тем, что узлы сообщаются через общую шину. Как видно из результатов для 1000 вершин, предположение относительно стандартных настроек планировщика оказалось верным.

Выводы

Реализация предлагаемой модели показала уровень производительности, близкий к ожидаемому, время работы составило примерно 3 мс для 500 вершин: отставание от идеализированного варианта значительно в процентном выражении, но мало в абсолютном, погрешность также вносит таймер. Проблемой реализации оказался стандартный планировщик Erlang. В дальнейшем предполагается провести серию опытов с измененными параметрами планировщика для определения оптимальных параметров размещения вершин по узлам вычислительной системы.

Список литературы

1. Shloegel K., Karypis G., Kumar V. Graph Partitioning for High Performance Scientific Simulations. Minneapolis: University of Minnesota, 2000. 40 p.
2. Гергель В.П. Теория и практика параллельных вычислений. М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2013. 423 с.
3. Чезарини Ф., Томпсон С. Программирование в Erlang. М.: Изд-во «ДМК», 2012. 488 с.
4. Овчинников В.А. Алгоритмизация комбинаторно-оптимизационных задач при проектировании ЭВМ и систем. М.: Изд-во МГТУ им. Н.Э.Баумана, 2001. 288 с.