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

УДК 519.876.5

Использование сетей Петри для поиска тупиков в вычислительных системах с общей памятью

***Сорокин Д.А.**, студент*
Россия, 105005, г. Москва, МГТУ им. Н. Э. Баумана,
кафедра «Программное обеспечение ЭВМ и информационные технологии»

Научный руководитель: Рудаков И.В., к.т.н., доцент, заведующий кафедрой
«Программное обеспечение ЭВМ и информационные технологии»
Россия, 105005, г. Москва, МГТУ им. Н. Э. Баумана
irudakov@bmstu.ru

Введение

Поскольку большинство современных вычислительных систем содержат несколько вычислительных компонент, то для таких систем зачастую разрабатывают приложения с использованием методик параллельного программирования, что позволяет таким приложениям выполнять несколько задач одновременно, используя все вычислительные возможности системы. Однако параллельные вычислительные системы по своей природе недетерминированы. Например, для одних и тех же входных данных параллельная программа может выдать либо различные, но верные результаты, либо выдать один и тот же результат, но в различном порядке [3].

Многопоточные программы сложнее разрабатывать и тестировать из-за недетерминированного порядка выполнения команд и синхронизации между задачами. Кроме того, многопоточные программы подвержены такому классу ошибок, как тупики, гонки, неправильная синхронизация и т. п.

Несмотря на многообразие существующих инструментов и подходов, отладка остается наиболее сложным и трудоемким этапом процесса разработки параллельных программного обеспечения.

Для тестирования и проверки многопоточных приложений применяют различные подходы. В данной статье для этого используется математический аппарат, называемый сетями Петри.

Классификация параллельных вычислительных систем

Существует много разновидностей параллельных вычислительных систем, различающихся своей архитектурой, степенью параллелизма программ, способом взаимодействия между отдельными вычислительными компонентами и другими характеристиками.

Для разбиения этого множества на отдельные классы систем было предложено немало классификаций [1]. В данной статье рассмотрена классификация, предложенная Е. Кришнамарфи (рис. 1). Он предложил для классификации параллельных вычислительных систем использовать следующие характеристики:

- Степень гранулярности.
- Способ реализации параллелизма.
- Топология и природа связи процессоров.
- Способ управления процессорами.

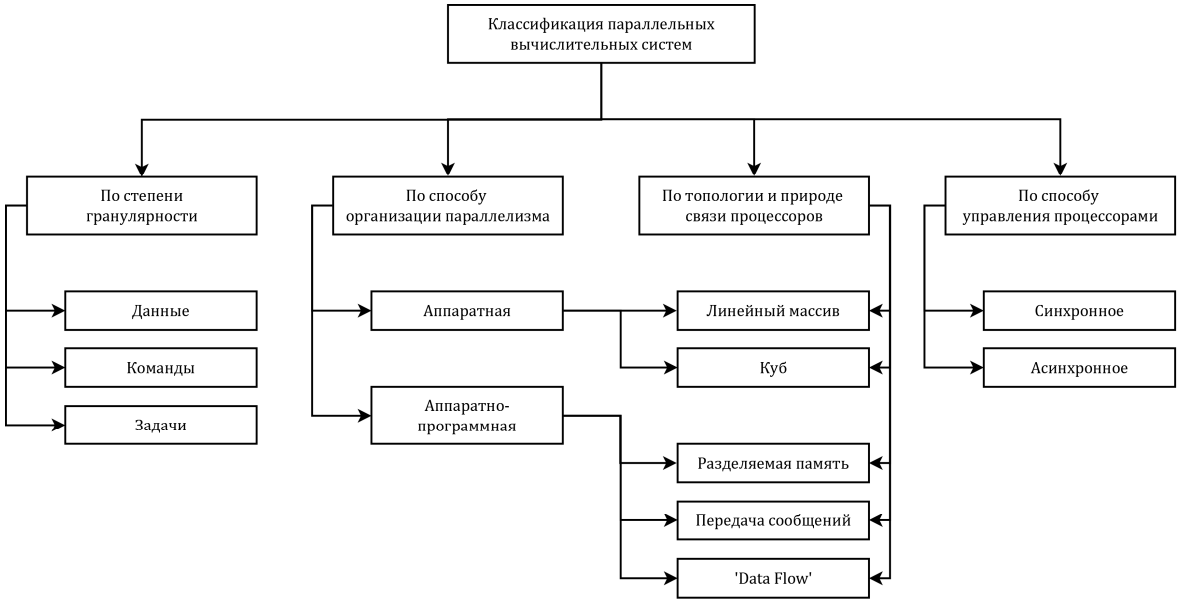

Рис. 1. Классификация параллельных вычислительных систем Е. Кришнамарфи

Первые две характеристики схожи с характеристиками, использованными в классификации Базу. Степень гранулярности определяет, какой уровень параллелизма используется в вычислительной системе. Если одна и та же операция может одновременно выполняться над некоторым набором данных, то данная вычислительная система

реализует параллелизм на уровне данных. Если вычислительная система способна выполнять более одной операции одновременно, то это указывает на параллелизм на уровне команд. Если же система спроектирована так, что целые последовательности инструкций могут быть выполнены одновременно, то это говорит о параллелизме на уровне задач.

Вторая характеристика классификации описывает метод реализации алгоритма. На текущий момент стало возможным реализовывать аппаратно не только простые математические операции, но и некоторые алгоритмы целиком, например, быстрое преобразование Фурье или операции над матрицами. Таким образом, способ организации параллелизма разделяет вычислительные системы на системы с аппаратной реализацией алгоритмов и на системы с традиционной программной реализацией алгоритмов.

Третья характеристика, - топология и природа связи процессоров, - напрямую связана со второй. Для аппаратного способа реализации параллелизма необходимо рассматривать топологию связи процессоров (матрица, линейный массив, …) и степень связанности процессоров между собой (сильная, средняя или слабая), а для аппаратно-программной реализации, помимо предыдущих характеристик, необходимо также рассмотреть механизм взаимодействия процессоров: через передачу сообщений, с помощью разделяемой памяти или по принципу *dataflow* (по готовности операндов вычислений).

Последняя четвертая характеристика указывает на способ управления процессорами и определяет общий принцип функционирования всей совокупности процессоров вычислительной системы: синхронный и асинхронный.

Для указания конкретных классов параллельных вычислительных систем следует скомбинировать все четыре характеристики. Так, например, векторно-конвейерные компьютеры описываются гранулярностью на уровне данных, аппаратной реализацией параллелизма, простой топологией процессоров со средней связанностью и синхронным способом управления [1].

В данной работе были рассмотрены классические мультипроцессоры (к ним относятся, например, широко распространенные персональные ЭВМ с многоядерными процессорами):

- Степень гранулярности — на уровне задач.
- Способ реализации параллелизма — аппаратно-программная.
- Топология и природа связи процессоров — простая топология со слабой связностью и использованием разделяемых переменных.

- Способ управления процессорами — асинхронный.

Одним из методов исследования параллельных вычислительных систем является их формализация в виде сетей Петри.

Сети Петри

Сеть Петри является формальной графической моделью для описания асинхронных параллельных процессов. Сеть представляет собой двудольный граф с некоторым начальным состоянием, называемым начальной маркировкой M_0. Вершины графа делятся на два типа: на вершины-позиции и вершины-переходы. Дуги, соединяющие вершины, не могут соединять вершины одного типа непосредственно; они соединяют переходы с позиции, а позиции — с переходами. В графическом представлении вершины-позиции отображаются в виде кругов, а вершины-переходы в виде прямоугольников. Дуги, соединяющие вершины, помечаются весами (целое положительное число) так, что дуга с весом N может быть рассмотрена как N параллельных дуг.

Маркировка сети сопоставляет каждой вершине-позиции неотрицательное целое число. Если маркировка задает для позиции p неотрицательное число k, то это значит, что позиция p содержит k фишек. Графически это отображается в виде k черных точек (фишек) внутри вершины p. Маркировка M задается вектором m, размер которого равен общему числу вершин-позиций. P-ая компонента этого вектора, обозначаемая как $M(p)$, указывает на число фишек в позиции p.

Таким образом, сети Петри можно представить в виде пятерки:

$$PN = (P, T, F, W, M_0),$$

где P – множество позиций, T – множество переходов, F – множество дуг между вершинами, W – весовая функция, и M_0 – изначальная маркировка:

$$P = \{p_1, p_2, \ldots, p_n\}$$

$$T = \{t_1, t_2, \ldots, t_m\}$$

$$F \subseteq (P \times T) \cup (T \times P)$$

$$W: F \rightarrow \{1, 2, \ldots\}$$

$$M_0: P \rightarrow \{1, 2, \ldots\}$$

$$P \cap T = \varnothing$$

$$P \cup T = \varnothing$$

Основные элементы сети Петри изображены на рисунке 2.

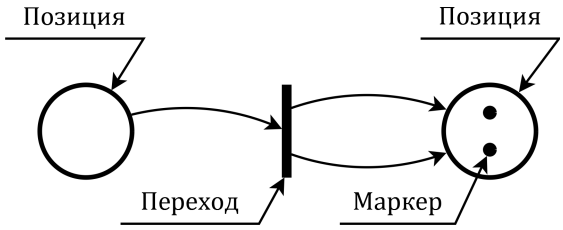

Рис. 2. Сеть Петри

Поведение многих параллельных систем может быть описано в терминах состояния системы и его изменения. Для того, чтобы смоделировать динамические характеристики системы, позиции и маркировка сети изменяется в соответствии со следующими правилами переходов [3]:

1) Переход *t* называется активным, если каждая входная позиция *p* перехода *t* помечена как минимум *w(p, t)* фишками, где *w(p, t)* – вес дуги, проведенной от позиции *p* до перехода *t*.
2) Активный переход может как сработать, так и не сработать.
3) Запуск активного перехода *t* удаляет *w(p, t)* фишек для каждой из входных позиций p и добавляет *w(p, t)* фишек для каждой из выходных позиций *p* перехода *t*. *w(t, p)* – вес дуги, проведенной от перехода *t* до позиции *p*.

Анализ модели программы

Для формализации параллельного приложения в виде сети Петри необходимо выполнить следующие действия:

1) Разработать модель алгоритма.
2) Перевести модель алгоритма на язык сетей Петри.
3) Проанализировать построенную модель Петри.

Сети Петри удачно представляют структуру управления программы. Переходы сети могут представлять вычисления, производимые программой, запуск функций и т. п., а позиции — передачу управления в программе. Фишкой удобно обозначить текущую инструкцию. По мере выполнения инструкций программы фишка продвигается по сети. Фишка, находящаяся в какой-либо позиции, указывает на готовность выполнения следующей инструкции.

Если какой-либо переход в сети Петри никогда не может быть запущен, такое состояние называется тупиком. Обнаружение возможности наступления состояния тупика

в сети Петри показывает наличие возможности состояния тупика и в моделируемой вычислительной среде.

Тупик может произойти, когда несколько потоков совместно используют некоторое множество ресурсов и операции внутри этих потоков требуют доступа более чем к одному из этих ресурсов, т. е. каждому из потоков необходимо заблокировать каждый из ресурсов перед выполнением операции. Если потоки используют одно и те же ресурсы, и последовательность их захвата непостоянна, то есть шанс оказаться в тупике.

Доступ к ресурсам зачастую реализуют в виде примитива типа *mutex,* который представляет собой средство взаимоисключения вычислительных потоков и имеет следующие методы: *lock* (ожидать освобождения с немедленным захватом) и *unlock* (освободить *mutex,* сделав возможным его захват другим потоком). Этот примитив можно представить в виде участка сети Петри (рис. 3).

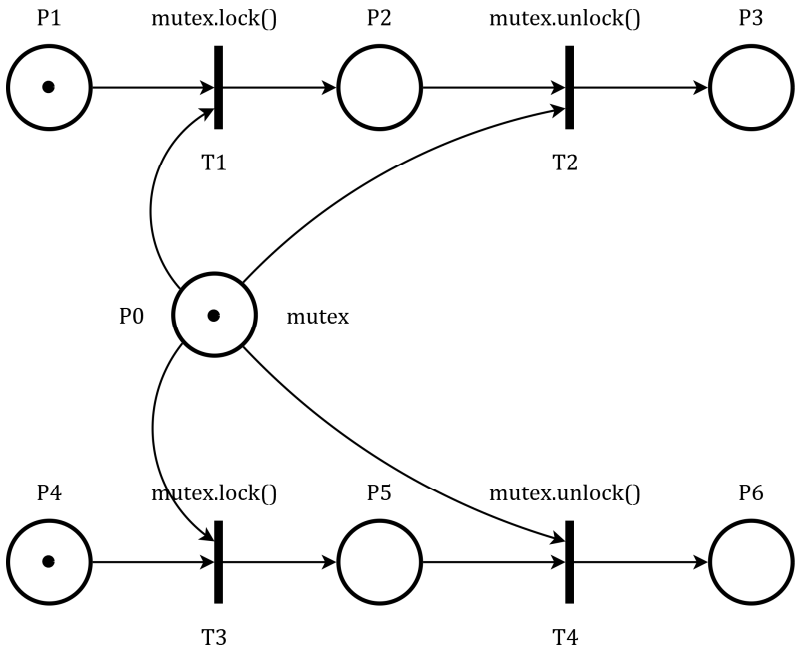

Рис. 3. Взаимодействие *mutex*-а и двух потоков

Позиция P_0 помечена фишкой в случае, если *mutex* свободен. Если же в позиции P_0 фишки нет, то это значит, что *mutex* уже занят и поток будет ожидать его освобождения. Переходы T_1 и T_3 соответствует вызову функции *lock()* и забирает фишку у позиции P_0. Переходы T_2 и T_4 соответствует вызову функции *unlock()* и возвращает фишку позиции P_0.

Позиции P_1, P_2, P_3 и переходы T_1, T_2 относятся к первому потоку, а позиции P_4, P_5, P_6 и переходы T_3, T_4, соответственно, ко второму потоку. В случае срабатывания перехода T_1 (т. е. захвата *mutex*-а) второй поток не сможет продолжить свое выполнение, т. к. переход T_3 не сможет сработать, пока не сработает переход T_2.

Для нахождения тупиковых маркировок в сети построим дерево достижимости сети [2]. Корневая вершина дерева представляет изначальную маркировку. Из каждой вершины могут исходить дуги, которые соответствуют срабатывающим переходам. Любой путь в дереве, начинающийся из его корня, описывает допустимую последовательность переходов. В случае, если число фишек в сети потенциально не ограничено, то дерево достижимости сети может быть бесконечным.

Для построения конечного дерева достижимости введем следующую классификацию маркировок:

1) Граничными являются маркировки, которые ещё не были обработаны алгоритмом. После обработки такие вершины переходят в один из других классов.

2) Терминальными являются маркировки, в которых нет разрешенных переходов.

3) Дублирующими являются маркировки, которые ранее уже встречались в дереве. Никакие производные из них маркировки рассматривать не нужно, так как они также будут присутствовать в дереве.

4) *Внутренними* являются обработанные, бывшие граничные маркировки, не являющиеся терминальными или дублирующими.

Для сведения дерева достижимости к конечному представлению сделаем следующее. Для позиций, которые увеличивают число фишек некоторой последовательностью запусков переходов, можно создать сколь угодно большое число фишек, просто повторяя эту последовательность. Бесконечное число маркировок такого типа обозначим как *w*, что обозначает «бесконечность». Таким образом, в маркировке число фишек может быть либо неотрицательным, либо *w*.

На первом шаге алгоритма начальная маркировка задается как корень дерева. Далее до тех пор, пока имеются граничные вершины, они обрабатываются алгоритмом. Пусть *x* – очередная граничная вершина. Тогда:

1) Если в дереве уже есть другая не граничная вершина *y* с такой же маркировкой *M(x) = M(y)*, то вершина *x* является дублирующей вершиной.

2) Если для вершины *x* ни один из переходов не активен, то x – терминальная вершина.

3) Иначе для каждого разрешенного перехода $t_i \in T$ необходимо создать вершину *z* дерева достижимости. Маркировка *M(z)* этой вершины определяется по следующим правилам:

a. Если *M(x) = w*, то и *M(z) = w*.

b. Если на пути от корня дерева к вершине *x* существует другая вершина *y* такая, что их маркировки отличаются числом фишек только для одной позиции, то *M(z) = w*.

c. Иначе *M(z)* принять как результат выполнения перехода t_i для маркировки *M(x)*.

4) Далее вершины *x* и *z* соединятся дугой t_j. Вершина *x* помечается как внутренняя, а вершина *z* – как граничная. Как только заканчиваются граничные вершины, алгоритм заканчивает свою работу.

Особенностью данного метода анализа параллельных приложений, формализованных сетью Петри, является необходимость отличать успешное завершение работы сети от тупиковой ситуации. Для этого рекомендуется ввести дополнительное конечное состояние сети [2].

Таким образом, сеть Петри не содержит тупиков, если для любой достижимой маркировки (кроме конечной) имеется по крайней мере один разрешённый переход.

Заключение

В статье показано использование сетей Петри для поиска тупиков в вычислительных системах с общей памятью. Приведена классификация параллельных вычислительных систем, описаны критерии различия таких систем. Описан математический аппарат сетей Петри, правила срабатывания переходов, а также алгоритм нахождения дерева достижимости для произвольной сети. Рассмотрены общие принципы формализации программ в виде сетей Петри. Указан метод поиска тупиков в параллельном приложении с использованием дерева достижимости сети.

Список литературы

1. Информационно-аналитический центр по параллельным вычислениям. Режим доступа: https://parallel.ru/ (дата обращения 25.11.2014).

2. Пащенкова А. В. Метод формализации программного обеспечения иерархическими сетями Петри // Молодежный научно-технический вестник. МГТУ им. Н.Э. Баумана. Электрон. журн. 2013. № 6. Режим доступа: http://sntbul.bmstu.ru/doc/577675.html (дата обращения 25.11.2014).
3. Kavi K. M., Moshtaghi A. R., Deng-Jyi C. Modeling Multithreaded Applications Using Petri Nets // International Journal of Parallel Programming. 2002. Vol. 30. Issue 5. P. 353-371. DOI: 10.1023/A:1019917329895.