|  |  |
| --- | --- |
| лого-РГСУ-2015.png | **Российский государственный социальный университет**  **Факультет информационных технологий** |

Практическое задание № 4

**по дисциплине «Технологии программирования»**

|  |  |
| --- | --- |
| **ФИО студента** | Тарасова Андрей Валерьевич |
| **Направление подготовки** | Программная инженерия |
| **Группа** | ПИН-2016 |

**Москва 2019**

*Параллельное программирование*

Понятие последовательного и параллельного алгоритма, последовательной и параллельной программы. Закон Амдаля. Цели создания параллельной программы

### Определение, назначение параллельного программирования

Существуют различные способы написания программ, которые условно можно разделить на три группы:

1. Последовательное программирование с дальнейшим автоматическим распараллеливанием.
2. Непосредственное формирование потоков параллельного управления, с учетом особенностей архитектур параллельных вычислительных систем или операционных систем.
3. Описание параллелизма без использования явного управления обеспечивается заданием только информационных связей. Предполагается, что программа будет выполняться на вычислительных системах с бесконечными ресурсами, операторы будут запускаться немедленно по готовности их исходных данных.

Каждый из перечисленных подходов обладает своими достоинствами и недостатками параллельное *программирование*.

***Параллельные вычисления*** - способ организации компьютерных вычислений, при котором программы разрабатываются, как набор взаимодействующих вычислительных процессов, работающих асинхронно и при этом одновременно.

***Параллельное программирование*** - это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (*multithreading*).

### Закон Амдала

Закон Амдала (1967 год), описывает максимальный теоретический выигрыш в производительности параллельного решения *по*отношению к лучшему последовательному решению. Закон Амдала описывается следующей математической формулой:

S_n=\frac{1}{\alpha+\frac {1-\alpha}{n}

где S_n - во сколько раз можно ускорить вычисления (ускорение), n - количество процессоров (ядер), \alpha - доля последовательно вычисляемого кода (\alpha \neq 0).

Закон Амдаля, несмотря на то, что он не учитывает многих факторов, накладывает ограничения на максимально достижимую эффективность параллельного алгоритма.

Предположим, например, что \alpha =\frac{1}{3}, то есть две трети операций в алгоритме могут выполняться параллельно, а треть - нет. Тогда ускоре-ние S_n<3. Таким образом, независимо от количества процессоров (ядер) и даже при игнорировании всех затрат на подготовку данных нельзя ускорить решение задачи более, чем в три раза.

Целью любой работы, связанной с параллельным программированием, является рассмотрение взаимосвязей между структурой математического алгоритма и архитектурой многопроцессорной вычислительной системы. В зависимости от сложности поставленной задачи возможна реализация различных видов взаимосвязей. Последние называются уровнями декомпозиции исходной задачи. Они могут быть определены следующим образом [31]:

Первый уровень - разбивка задачи на подзадачи.

Второй уровень - разбиение каждой отдельно взятой подзадачи на множество одновременно выполняемых при различных исходных данных квазиоднотипных процедур. В математической физике такой тип параллелизма называют геометрическим параллелизмом или параллелизмом данных, так как распараллеливание здесь производится путем распределения вычислений в разных точках расчетной области в разные процессоры.

Третий уровень - распараллеливание отдельных процедур.

Наиболее глубокий уровень распараллеливания, четвертый - разбиение арифметических процессов на число процессоров.

Последний не рекомендуется к использованию на суперЭВМ с распределенной памятью, у которых на каждый процессор выделена локальная память. Исследователями большинства прикладных задач рекомендуется останавливать процесс их декомпозиции на втором уровне.

Способы и средства разработки параллельных программ (общая характеристика)

В общем плане под параллельными вычислениями понимаются процессы обработки данных, в которых одновременно могут выполняться несколько операций компьютерной системы. Достижение параллелизма возможно только при выполнении следующих требований к архитектурным принципам построения вычислительной среды:

\* независимость функционирования отдельных устройств ЭВМ – данное требование относится в равной степени ко всем основным компонентам вычислительной системы: к устройствам ввода-вывода, обрабатывающим процессорам и устройствам памяти;

\* избыточность элементов вычислительной системы – организация избыточности может осуществляться в следующих основных формах:

- использование специализированных устройств, таких, например, как отдельные процессоры для целочисленной и вещественной арифметики, устройства многоуровневой памяти (регистры, кэш);

- дублирование устройств ЭВМ путем использования, например, нескольких однотипных обрабатывающих процессоров или нескольких устройств оперативной памяти.

Дополнительной формой обеспечения параллелизма может служить конвейерная реализация обрабатывающих устройств, при которой выполнение операций в устройствах представляется в виде исполнения последовательности составляющих операцию подкоманд. Как результат, при вычислениях на таких устройствах на разных стадиях обработки могут находиться одновременно несколько различных элементов данных.

При рассмотрении проблемы организации параллельных вычислений следует различать следующие возможные режимы выполнения независимых частей программы:

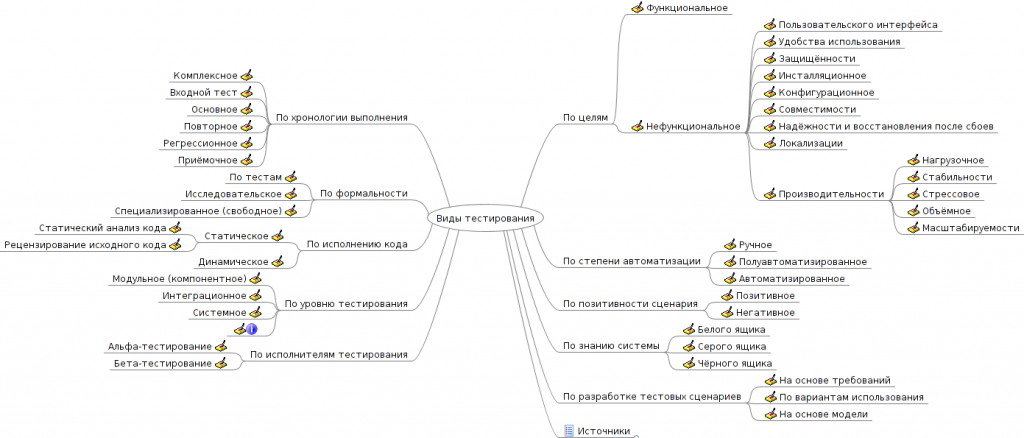
\* многозадачный режим (режим разделения времени), при котором для выполнения нескольких процессов используется единственный процессор. Данный режим является псевдопараллельным, когда активным (исполняемым) может быть один, единственный процесс, а все остальные процессы находятся в состоянии ожидания своей очереди; применение режима разделения времени может повысить эффективность организации вычислений (например, если один из процессов не может выполняться из-за ожидания вводимых данных, процессор может быть задействован для выполнения другого, готового к исполнению процесса). Кроме того, в данном режиме проявляются многие эффекты параллельных вычислений (необходимость взаимоисключения и синхронизации процессов и др.), и, как результат, этот режим может быть использован при начальной подготовке параллельных программ;

\* параллельное выполнение, когда в один и тот же момент может выполняться несколько команд обработки данных. Такой режим вычислений может быть обеспечен не только при наличии нескольких процессоров, но и при помощи конвейерных и векторных обрабатывающих устройств;

\* распределенные вычисления; данный термин обычно применяют для указания параллельной обработки данных, при которой используется несколько обрабатывающих устройств, достаточно удаленных друг от друга, в которых передача данных по линиям связи приводит к существенным временным задержкам. Как результат, эффективная обработка данных при таком способе организации вычислений возможна только для параллельных алгоритмов с низкой интенсивностью потоков межпроцессорных передач данных. Перечисленные условия являются характерными, например, при организации вычислений в многомашинных вычислительных комплексах, образуемых объединением нескольких отдельных ЭВМ с помощью каналов связи локальных или глобальных информационных сетей.

В дальнейшем основное внимание будет уделяться второму типу организац

*Тестирование программ*

Классификации видов и методов тестирования

Проектирование тестов больших программ.

Проектирование тестов больших программ пока в большей мере остается искусством и в меньшей мере является наукой. Чтобы построить разумную стратегию тестирования, надо разумно сочетать оба этих два крайних подхода и пользоваться математическими доказательствами.

**Восходящее тестирование.**Сначала автономно тестируются модули нижних уровней, которые не вызывают других модулей. При этом достигается такая же их высокая надежность, как и у встроенных в компилятор функций. Затем тестируются модули более высоких уровней вместе с уже проверенными модулями и т. д. по схеме иерархии.

При восходящем тестировании для каждого модуля необходима ведущая программа. Это монитор или драйвер, который подает тесты в соответствии со спецификациями тестов. Ряд фирм выпускает промышленные драйверы или мониторы тестов.

Нисходящее тестирование. При этом подходе изолированно тестируется головной модуль или группа модулей головного ядра. Программа собирается и тестируется сверху вниз. Недостающие модули заменяются заглушками.

*Достоинства*нисходящего тестирования: этот метод совмещает тестирование модуля с тестированием сопряжений и частично тестирует функции модуля. Когда уже начинает работать ввод/вывод модуля, удобно готовить тесты.

*Недостатки*нисходящего тестирования: модуль редко досконально тестируется сразу после его подключения. Для основательного тестирования требуются изощренные заглушки. Часто программисты откладывают тщательное тестирование и даже забывают о нем. Другой недостаток — желание начать программирование еще до конца проектирования. Если ядро уже запрограммировано, то возникает сопротивление всяким его изменениям, даже для улучшения структуры программы. В конечном итоге, именно рационализация структуры программы за счет проведения проектных итераций способствует достижению большей экономии, чем даст раннее программирование.

*Модифицированный нисходящий метод.*Согласно этому методу, каждый модуль автономно тестируется перед включением в программу, собираемую сверху вниз.

*Метод большого скачка*— каждый модуль тестируется автономно. По окончании автономного тестирования всех модулей модули просто интегрируются в готовую программную систему. Как правило, этот метод нежелателен. Однако если программа мала и хорошо спроектирована по сопряжениям, то метод большого скачка вполне приемлем.

*Метод сандвича*представляет собой компромисс между нисходящим и восходящим подходами. По этому методу реализация и тестирование ведутся одновременно сверху и снизу, и два этих процесса встречаются в заранее намеченной временной точке.

*Модифицированный метод сандвича:*нижние модули тестируются строго снизу вверх, а модули верхних модулей сначала тестируются автономно, а затем собираются нисходящим методом.

Разработка через тестирование

Методология разработки через тестирование, или TDD (Test Driven Development), появилась в нашей отрасли уже более 10 лет. Изначально она применялась на волне экстремального программирования (XP, eXtreme Programming), но с тех пор была принята на вооружение Scrum и практически всеми остальными гибкими (Agile) методологиями. Даже группы, не использующие гибкие методологии, применяют TDD.

Когда в 1998 году я впервые услышал о «упреждающем тестировании», я отнесся к нему скептически. Да и кто бы поступил иначе? Кто *начинает*работу с написания модульных тестов? Кто будет делать подобные глупости?

Но к тому времени у меня был уже 30-летний опыт профессионального программирования; я видел, как в отрасли появляются и исчезают новые идеи. Я прекрасно понимал, что ничего не стоит отвергать заранее, особенно если рекомендует такой человек, как Кент Бек.

Так в 1999 году я отправился в Медфорд, штат Орегон, чтобы встретиться с Кентом и научиться у него новой методологии. Результат был просто поразительным!

Мы с Кентом сели у него в офисе и начали программировать простую задачу на Java. Я хотел просто написать свой примитивный код, но Кент воспротивился и провел меня по всему процессу шаг за шагом. Сначала он написал крошечную часть модульного теста, которую и кодом-то нельзя было назвать. Затем он написал код, достаточный для того, чтобы тест компилировался. Затем он написал еще один тест и еще немного кода.

Такой рабочий цикл полностью противоречил всему моему опыту. Я привык писать код не менее часа, прежде чем пытаться откомпилировать или запустить его. Но Кент буквально выполнял свой код каждые 30 секунд или около того. Это было невероятно! Но самое интересное, что этот рабочий цикл был мне знаком! Я сталкивался с ним много лет назад, когда еще ребенком[19] программировал игры на интерпретируемых языках вроде Basic или Logo. В этих языках не было сборки как таковой: вы просто добавляли строку кода и запускали программу. Рабочий цикл проходил очень быстро. И по этой причине программирование на этих языках бывало *очень*производительным.

Но в *настоящем*программировании такой рабочий цикл казался абсурдным. В настоящем программировании вы тратили много времени на написание кода, а потом еще больше времени на то, чтобы заставить его компилироваться. И еще больше времени на отладку. Я ведь *был программистом C++, черт побери*! А в C++ процессы сборки и компоновки могли длиться минутами, а то и часами. Тридцатисекундные рабочие циклы казались немыслимыми. Тем не менее передо мной сидел Кент, который писал свою программу на Java с 30-секундными циклами – и без малейшего намека на то, что работа замедлится. И тогда до меня дошло, что эта простая методология позволяет программировать на настоящих языках с продолжительностью рабочего цикла, типичной для Logo! И я капитально «подсел» на TDD!