Государственное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н. Э. Баумана»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Р А С Ч Ё Т Н О - П О Я С Н И Т Е Л Ь Н А Я З А П И С К А

к курсовой работе на тему:

Реализации инструмента анализа параллельных программ

Студент Капустин А.И. И.О. Фамилия
(Подпись, дата)

Руководитель курсового проекта Ковтушенко А.П. И.О. Фамилия
(Подпись, дата)

Москва 2017

Содержание

Введение

Вычислительные системы с массовым параллелизмом совершили революцию в мире параллельных вычислений, обеспечив производительность порядка нескольких Терафлоп/с. Необходимость в вычислениях такого объёма возникает в широком круге областей от молекулярной физики до предсказания погоды. На сегодняшний день массивно параллельные системы являются лидерами по критерию цена/производительность. Однако, за таким прогрессом аппаратного обеспечения не поспевало программное обеспечение. Очевидно, что будут созданы более мощные массивно параллельные системы, и основной проблемой становится разработка программного обеспечения, способного максимально использовать вычислительные ресурсы. Параллельные вычисления связаны с очень сложным и тяжело понимаемым ходом выполнения программ, и недостаточно полное понимание не позволяет полностью использовать вычислительные ресурсы. Так как производительность – самый важный критерий при использовании параллельных компьютеров, и так как производительность пилотных реализаций параллельных программ как правило далека от ожидаемой, модификация программы с целью повышения производительности является важной частью процесса разработки. Основным фактором, ведущим к большому количеству трудоёмкой работы, требующей высокой квалификации является нехватка средств для анализа и оценки производительности. При отсутствии таких средств, причины низкой производительности выявляются с помощью средств, разработанных для конкретного приложения и метода тыка. Повышение производительности параллельных программ наиболее важная, но трудоёмкая задача, которая требует хорошего понимания задачи. Цель средств контроля и визуализации производительности параллельных программ – помочь разработчику обрести это понимание. Визуализация производительности заключается в использовании графических образов для представления анализа данных о выполнении программы для лучшего её понимания. Такие средства визуализации были очень полезны и в прошлом, и широко используются в настоящее время для повышения производительности параллельных программ. Несмотря на прогресс в области визуализации научных данных, технологии визуализации для параллельного программирования являются объектом внимания многих разработчиков, так как требуется тем более сложная визуализация, чем более сложными становятся параллельные системы.

1 Аналитический раздел

Алгоритмы, с которыми приходится иметь дело при параллельном программировании гораздо сложнее, чем аналогичные последовательные алгоритмы. Часть проблем касается распараллеливания самого алгоритма (например, как разбить процесс на задачи и как поставить им в соответствие физические процессоры), часть касается оптимального использования вычислительных ресурсов (баланс загрузки процессоров, межпроцессные взаимодействия, доступ к кэшу). Для разработки высокопроизводительных приложений программисту не только очень хорошо знать структуру самого приложения, но и структуру аппаратной платформы, а зачастую и программной (особенности операционной системы, компилятора и.т.д.). Итак, уровень знаний, необходимый для разработки высокопроизводительных параллельных программ выше, чем для последовательных программ. Параллельное программирование может быть очень мучительным и запутанным. В добавок отладка параллельной программы и поиск критических мест также трудоёмкий и длительный процесс. Из-за этих причин программирование для массивно параллельных машин считается непростым делом, и в значительной степени причиной этому служит недостаток инструментария по сравнению с последовательными машинами. Для того, чтобы понять поведение и внутреннюю логику параллельных программ и помочь программисту выявить потенциальные слабые места (например, ограничения по межпроцессным взаимодействиям) необходимы средства контроля хода выполнения параллельной программы. Они могут использоваться для сравнения производительности схожих программ или помогать при отладке.

1.1 Парадигмы параллельного программирования

1.1.1 Структура и характеристики параллельных алгоритмов

Любой параллельный алгоритм (программа) состоит из блоков последовательных и параллельных вычислений. Последовательная часть программы (называется критическим сечением) – это последовательность операторов, которая должна выполняться только одним процессором. За критическим сечением обычно следует ветвление, инициирующее параллельно выполняемые участки программы (параллельные сегменты). В месте соединения параллельных сегментов выполняется синхронизация и параллельные сегменты возвращаются к критическому сечению. Синхронизация требуется для того, чтобы вычисления в параллельных сегментах закончились прежде, чем начнется выполнение последовательной части.

1.1.2 Программные парадигмы

При параллельном программировании появляются дополнительные сложности (по сравнению с последовательным программированием): возникает необходи-

мость управлять несколькими процессорами и координировать межпроцессорные вызовы. В общем случае параллельная программа представляет собой ряд заданий, работающих параллельно друг другу и взаимодействующих между собой. Существует несколько парадигм (моделей) программирования при создании параллельных алгоритмов. Наиболее распространенными парадигмами являются модель передачи сообщений и модель разделяемой памяти.

1.1.3 Проектирование параллельных алгоритмов

Несмотря на то, что проектирование параллельных алгоритмов есть процесс творческий и в общем случае достаточно сложно рекомендовать универсальный рецепт, следует все же привести общую методологию проектирования параллельных алгоритмов, которая в значительной степени оградит проектировщика от потенциальных ошибок.

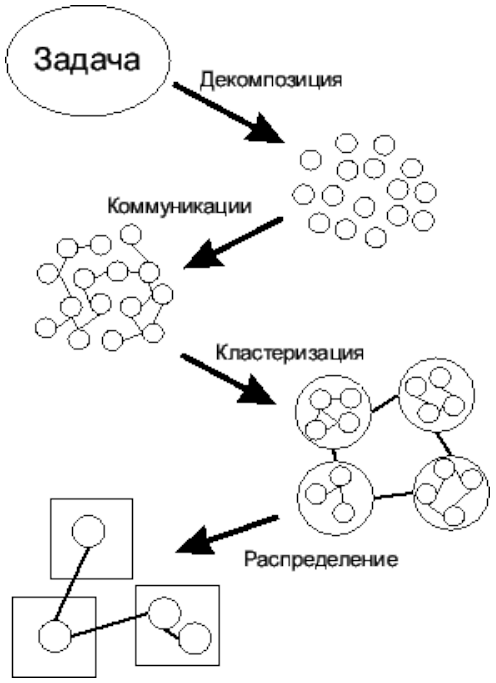

Рисунок 1.1 — Методология проектирования параллельных алгоритмов

Описываемая методология проектирования предлагает подход к распараллеливанию, который рассматривает машинно-независимые аспекты реализации алгоритма, такие как параллелизм, на первой стадии, а особенности проектирования, связанные с конкретным параллельным компьютером, - на второй. Данный подход разделяет процесс проектирования на четыре отдельных этапа: декомпозиция (partitioning), коммуникации (communications), кластеризация (agglomeration) и распределение (mapping). Первые два этапа призваны выделить в исходной задаче параллелизм и масштабируемость, остальные этапы связаны с различными аспектами

производительности алгоритма[1]. Рисунок 1.1 иллюстрирует процесс проектирования параллельного алгоритма. Сформулируем кратко содержание этих этапов.

1.1.3.1 Декомпозиция

Общая задача вычислений и обработки данных делится на меньшего размера подзадачи. При этом игнорируются проблемы практической реализации, например, число процессоров используемого в будущем компьютера. Напротив, все внимание сосредотачивается на возможном параллелизме исходной задачи.

1.1.3.2 Коммуникации

Определяется требуемая связь между подзадачами, ее структура и конкретные алгоритмы коммуникаций.

1.1.3.3 Кластеризация

Структура подзадач и коммуникаций оценивается с учетом требований производительности алгоритма и затрат на реализацию. Если необходимо, отдельные подзадачи комбинируются в более крупные с целью улучшения производительности и снижения затрат на разработку.

1.1.3.4 Распределение

Каждая подзадача назначается определенному процессору, при этом стараются максимально равномерно загрузить процессоры и минимизировать коммуникации. Распределение может быть статическим или формироваться в процессе выполнения программы на основе алгоритмов балансировки загрузки (load balancing algorithms). Следует заметить, что при прохождении последних двух этапов необходимо учитывать архитектуру параллельного компьютера, имеющегося в распоряжении разработчика, другими словами параллельный алгоритм должен быть адекватен используемой параллельной системе.

1.1.4 Характеристики производительности параллельного алгоритма

Задачей программиста является проектирование и реализация программ, которые удовлетворяют требованиям пользователя в смысле производительности и корректной работы. Производительность является достаточно сложным и многосторонним понятием. Кроме времени выполнения программы и масштабируемости следует также анализировать механизмы, отвечающие за генерацию, хранение и передачу данных по сети, оценивать затраты, связанные с проектированием, реализацией, эксплуатацией и сопровождением программного обеспечения. Существуют весьма раз-

нообразные параметры, влияющие на производительность вычислительной системы. Среди них наиболее значимыми являются: требования по памяти, производительность сети, время задержки при передаче данных (latency time), переносимость, масштабируемость, затраты на проектирование, реализацию, отладку, требование к аппаратному обеспечению и т.д. Критериями, по которым оценивается производительность, являются время выполнения программы, ускорение и эффективность. Относительная важность различных параметров будет меняться в зависимости от природы вычислительной системы и поставленной задачи. Специфика конкретной задачи может требовать высоких показателей для каких-либо определенных критериев, в то время как остальные должны быть оптимизированы или полностью игнорированы. Ниже мы рассмотрим лишь две характеристики производительности, а именно, эффективность и ускорение, поскольку они наиболее часто используются при анализе производительности параллельных алгоритмов.

1.1.4.1 Ускорение и эффективность

Ускорение параллельного алгоритма S_p определяется следующим образом:

$$S_{p0} = \frac{T_0}{T_p} \tag{1.1}$$

$$S_p = \frac{T_1}{T_p} \tag{1.2}$$

где T_0- время работы наилучшего последовательного алгоритма, T_1 – время работы параллельного алгоритма на одном процессоре, T_p – время работы параллельного алгоритма на p процессорах.Следует отметить, что в общем случае T0<T1, т.е. последовательный алгоритм работает быстрее, чем параллельный алгоритм с использованием одного процессора, т.к. в любом параллельном алгоритме всегда присутствует дополнительная часть кода, обеспечивающая распараллеливание вычислений, а именно точки синхронизации, передача данных от одного процессора к другому и т.д. Кроме того, наиболее оптимальные алгоритмические решения не всегда хорошо параллелизуются. Таким образом, при анализе ускорения алгоритма следует всегда оговаривать, какое из определений используется [2]. В общем случае редко удается полностью распараллелить вычислительный процесс и получить максимальное ускорение в p раз. и один параллельный алгоритм не может обеспечить идеальную балансировку нагрузки процессоров, т.е. в равной степени разделить вычислительную работу между процессорами. Эффективность E_p параллельного алгоритма характеризует степень загрузки процессоров и определяется как отношение ускорения S_p к максимально возможному ускорению p:

$$E_p = \frac{S_p}{p} \tag{1.3}$$

Нетрудно заметить, что

$$1 \leqslant S_p \leqslant p, \ \frac{1}{p} \leqslant E_p \leqslant 1 \tag{1.4}$$

1.1.4.2 Закон Амдала

Пусть $\alpha \leqslant 1$ есть некая доля вычислений, которые выполняются в параллельном режиме, а доля $1-\alpha$ – в последовательном без совмещения этих режимов во времени. Тогда последовательные вычисления занимают время $(1-\alpha)T$, а параллельные – время $\frac{\alpha T_1}{p}$. Оба режима занимают время $T_p T_1(1-\alpha+\frac{\alpha}{p})$. Тогда формулы для ускорения S_p (1.2) и эффективности E_p (1.3) принимают вид:

$$S_p = \frac{p}{p-\alpha(p-1)} \tag{1.5}$$

$$E_p = \frac{1-\alpha}{(1-\alpha)p+\alpha} \tag{1.6}$$

Формулы (1.5) и (1.6) и выражают простую и обладающую большой общностью зависимость, известную как закон Амдала. Согласно этому закону в алгоритме, имеющем два не совмещенных во времени режима последовательных и параллельных вычислений, при $n \to \infty$ границу ускорения составляет величина, обратная доле последовательных вычислений:

$$\lim_{x\to\infty}(S_p) = \frac{1}{1-\alpha} \tag{1.7}$$

Для повышения скорости вычислений за счет распараллеливания требуется повы-

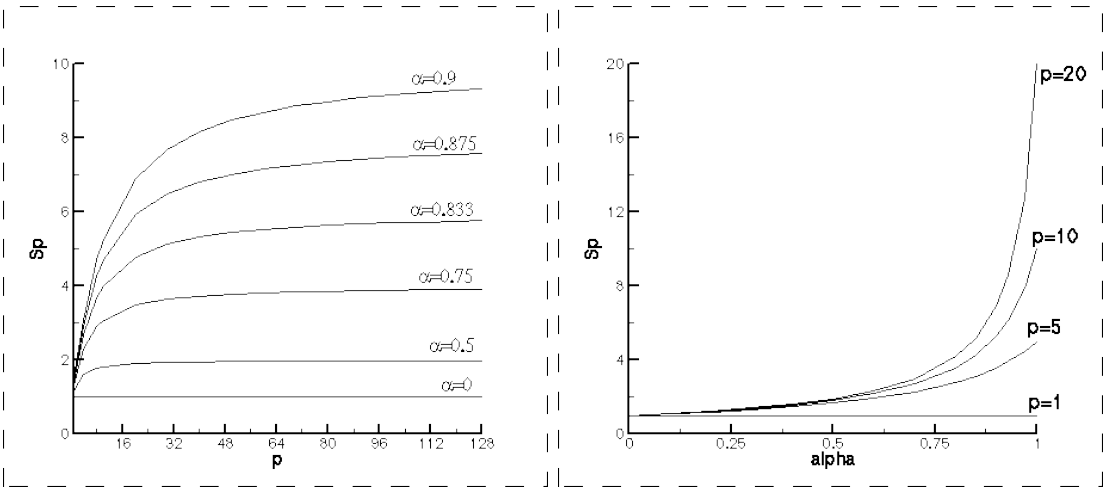

Рисунок 1.2 — Зависимость ускорения S_p от числа процессоров p для различных объемов параллельных вычислений α (слева) и от объема параллельных вычислений(α) для различного числа процессоров p (справа).

шать скорость параллельных вычислений, однако остающиеся последовательные вычисления все в большей степени тормозят ускорение в целом. Даже небольшая доля последовательных вычислений может сильно снизить общую скорость. Особенно

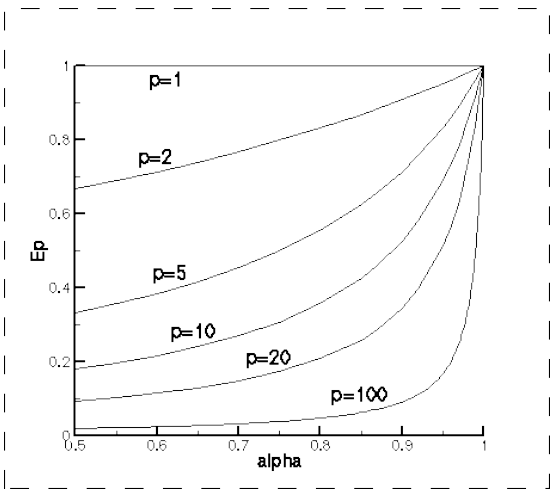

Рисунок 1.3 — Зависимость эффективности E_pот объема параллельных вычислений α для различного числа процессоров p.

сильное влияние на ускорение и эффективность алгоритма последовательные вычисления оказывают в области α , близкому к 1. Поэтому в алгоритмах с большим объемом параллельных вычислений заметным является уменьшение последовательных вычислений даже на доли процента от общего объема вычислений. Закон Амдала представляет собой приближенную модель ускорения, т.к. не учитывает так называемые накладные расходы и зависимость α от p. Однако в общем случае доля параллельных вычислений может меняться с изменением числа используемых процессоров (проблема масштабируемости).

1.1.4.3 Факторы, снижающие ускорение

Факторы, снижающие ускорение (overheads) К основным факторам, приводящим к снижению ускорения, относятся: 1. Разбалансировка нагрузки процессоров (load unbalancing). Неравномерная распределение вычислительной работы между процессорами - отдельные процессоры могут простаивать (idle processors) - не обеспечивает высокой степени параллелизма, что в свою очередь снижает эффективность; 2. Синхронизация процессоров (processor synchronization). На выполнение программного кода, обеспечивающего синхронизацию типа барьер, требуются временные затраты. 3. Коммуникации (communications). При обмене сообщениями между процессорами могут возникать задержки как вследствие особенностей программной реализации алгоритма (например, на момент принятия сообщения процессором оно еще не отправлено другим процессором), так и вследствие аппаратных причин (например, загруженность сети). 4. Параллельный ввод/вывод (parallel input/output). Например, программы, работащие на различных процессорах, одновременно выводят данные в один и тот же файл и т.д.

1.1.5 Наблюдение за выполнением параллельной программы

Практика показывает, что производительность параллельной программы далеко не всегда растёт с ростом мощности используемого аппаратного обеспечения. Причин этому много: пользователи хотят видеть более удобные сервисы системы, потребляющие вычислительную мощь, механизмы обеспечения безопасности также требуют процессорного времени. От этих эффектов невозможно избавиться. Со стороны разработчика эффективность использования вычислительных ресурсов может быть увеличена за счёт тщательной разработки программных продуктов с одной стороны и анализа поведения программ во время выполнения – с другой. Если анализ производительности является важным для последовательных программ, то для параллельных и распределённых систем он жизненно необходим. Причиной этому служат сложные взаимодействия между разными частями программы, работающими над решением одной общей задачи. Необходимость использования общих ресурсов ведёт к проблемам связанными с синхронизацией процессов, временем ожидания, тупиками и т.д. Кроме обычных проблем последовательной программы возникают новые трудности, связанные с управлением параллельными задачами, которые влекут к потере процессорной мощности, тем самым процессорная мощность не используется на том уровне, на котором могла. Наблюдение за ходом выполнения программы, с помощью событийного подхода, ведёт к большим объёмам файлов трасс. Для того чтобы сконцентрироваться на местах трассы, представляющих интерес, необходимо выявить их местоположение. Следовательно, для общего и быстрого анализа необходимо использовать статистические методы. После получения такой информации, можно применять более детальный подход. При этом частой задачей является не только измерение производительности системы или её части, но и получение представления о динамическом взаимодействии процессов.

1.1.5.1 Проблемы производительности параллельных систем

Настройка последовательных программ относительно проста: используется профилирование для определения наиболее часто используемых частей. Обычно, это лишь малая часть всего кода. Переработка этих частей программы приводит к увеличению производительности. Этот простой подход не работает в случае параллельных и распределённых программ. Например, оптимизация части программы, которая должна ждать промежуточного результата счёта другого процесса, не принесёт выгоды – этот процесс просто будет находиться в состоянии ожидания больше времени. Для того, чтобы определить почему программа ведёт себя так, как она себя ведёт, надо искать причины такого поведения. Это ведёт к рассмотрению причинно-следственных связей в системе. Как будет показано далее, анализ параллельной системы с этой токи зрения может привести к интересным результатам.

1.1.5.2 Детерминированность компьютерных системы

В общих чертах, детерминированность представляет собой закон, по которому определённое действие ведет к определённому результату. В контексте вычислительных систем детерминированность означает, что поведение процессов определяется законом, выраженным в виде программы. То есть будущее каждого процесса зависит от текущего положения в программе, текущего контекста относительно других процессов и следующей инструкции программы. Есть несколько специфичных вопросов, относящихся к параллельным системам. В последовательных программах инструкции выполняются в том же порядке, как и в тексте программы, за исключением ветвлений. В этом случае следующая инструкция определяется исходя их некоторых условий. Рассматривая все инструкции как потенциальные события, каждая не управляющая инструкция является причиной последующей инструкции, а управляющая инструкция может являться причиной одной из множества возможных инструкций. В то время, как причинно-следственные связи в последовательной программе очевидны, поскольку каждая инструкция является предпосылкой следующей инструкции, это не так для взаимодействующих процессов, которые обмениваются информацией для решения общей задачи. Обмен данными может состоять, например, из обмена частичными результатами примитивов синхронизации, барьеров. В параллельных системах каждый процесс имеет свой поток управления, но в добавление к нему, будущее состояние процесса зависит также от информации, полученной от других процессов. Передача этой информации посредством системных средств связи приводит к зависимости между процессами. В результате есть как пары событий в разных процессах как зависящие друг от друга, так и не зависящие. Зависимость между двумя событиями может быть вызвана двумя причинами: 1) события принадлежат одному процессу; 2) события включают коммуникацию (например, принимающее и отправляющее сообщение или записывающее и считывающее информацию в/из разделяемой памяти).

1.1.5.3 Анализ зависимостей

Рис. 1.4 показывает пример, когда три процесса A, B и C работают над одной задачей. Горизонтальными линиями обозначен ход выполнения одного процесса, точки показывают важные события, и стрелками показаны причинно-следственные связи. Такая связь автоматически подразумевает временнОю предшествование одного события другому.

Следуя по стрелкам от события к событию с учётом взаимодействий, мы приходим к событиям, зависящим от начального. В отличии от последовательной программы, не все события в будущем могут быть достигнуты с помощью такого обхода, т.е. не все события причинно-зависимы. Рассмотрим событие b2. Все собы-

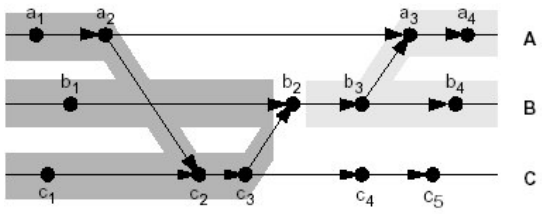

Рисунок 1.4 — Структура зависимостей событий

тия в светло-серой области зависят от b2, но c4 и c5 не могут быть достигнуты из b2, так они не зависят от b2. Все события в тёмно-серой области влияют на b2, но все события, следующие за a2 и c3 могут быть отброшены при поиске причин b2. При анализе трасс процессов с помощью таких зависимостей следует учесть, что: 1. необходимо знать положение каждого события локально внутри процесса; 2. должно быть возможно определить связанные пары событий в разных процессах Эти требования могут быть удовлетворены, если исследуемая система предоставляет соответствующую информацию о событиях с сохранением её в трассе. При обработке таких трасс достаточным критерием для соответствия двух событий одному и тому же сообщению является одинаковое количество «пишущих» событий в одном процессе и «читающих» - в другом. В других случаях, например, при перекрывающихся сообщениях, необходимо фиксировать дополнительную информацию, которая может быть использована для сопоставления пар событий. Примером такой информации могут служить номера пакетов в коммуникационных протоколах. Анализ зависимостей в компьютерных системах означает следование по связанным событиям из конкретной точки, которая представляет собой интересное (нежелательное) поведение системы. Все инструкции программы, являющиеся причиной данного события являются кандидатами на оптимизацию. В параллельных системах с большим количеством процессов такое ограничение по предшествующим событиям позволяет сконцентрироваться на основных моментах, что приводит к значительному снижению затрат на анализ.

1.1.5.4 Сбор данных

Учитывая сложность реальной вычислительной системы, не представляется возможным предсказать поведение системы во времени с помощью теоретических изысканий. По этой причине в большинстве инструментов программирования отсутствуют средства «планирования производительности». А те инструменты, в которые включён анализ производительности, дают лучшие результаты при профилировании. Профилирование оказывает большую помощь при программировании последовательных программ, при этом используется подход, основанный на измерениях. Т.е. определяется момент входа и выхода из каждой подпрограммы, подсчитывается количество вызовов каждой подпрограммы. Мониторинг(непрерывный процесс наблюдения и регистрации параметров объек) делает шаг вперёд по отношению к

простому измерению – маркеры событий вставляются в программный код в местах, предоставляющих определённый интерес, и достижение маркера может означать, что программа совершила определённые действия. С помощью этих средств программа может быть анализирована на разных уровнях абстракции, определяемых разметкой исходников.

1.1.5.5 Время сбора данных

Есть два разных подхода к определению моментов, когда информация о состоянии системы должна быть собрана профилировщиком. Традиционные методы используют временнОй подход, при котором информация непрерывно пишется по протоколу, схожему с медицинским оборудованием, или данные собираются с определённой частотой. Этот подход имеет право на жизнь при малой частоте сбора данных для получения общего вида поведения системы. Для получения информации о внутреннем поведении вычислительной системы необходима высокая частота сбора событий с одной стороны и длительный промежуток времени – с другой. При описанном подходе это приведёт к огромному потоку данных. Событийный мониторинг позволяет уменьшить объём данных на несколько порядков. При этом на важных частях программы фиксируется больше событий, чем на остальных. Тем самым достигается необходимая пользователю картина о динамическом поведении системы. При этом подходе измерения выполняемой программы порождают компактный вид системы, отбрасывающий детали, ненужные для решения текущего вопроса.

1.1.6 Технологии параллельного программирования

Широкое распространение компьютеров распределённой памятью определило и появление соответствующих систем программирования. Как правило, в таких системах отсутствует единое адресное пространство, и для обмена данными между параллельными процессами используется явная передача сообщений через коммуникационную среду. Отдельные процессы описываются с помощью традиционных языков программирования, а для организации их взаимодействия вводятся дополнительные функции. По этой причине практически все системы программирования, основанные на явной передачи сообщений, существуют не только в виде новых языков, а в виде интерфейсов и библиотек. К настоящему времени примеров известных систем программирования на основе передачи сообщений накопилось довольно много: Shmem, Linda, PVM, MPI, PARMACS, P4, MPL, NX и др. В этом разделе рассмотрим наиболее популярные системы.

1.1.6.1 Linda

Идея построения системы Linda исключительно проста, а потому красива и очень привлекательна. Параллельная программа есть множество параллельных процессов, и каждый процесс работает согласно обычной последовательной программе. Все процессы имеют доступ к общей памяти, единицей хранения в которой является кортеж. Отсюда происходит и специальное название для общей памяти - пространство кортежей. Каждый кортеж это упорядоченная последовательность значений.

1.1.6.2 PVM (Parallel Virtual Machine)

Параллельную виртуальную машину (PVM) можно определить как часть средств реального вычислительного комплекса (процессоры, память, периферийные устройства и т.д.), предназначенную для выполнения множества задач, участвующих в получении общего результата вычислений. В общем случае число задач может превосходить число процессоров, включенных в PVM.

1.1.6.3 MPI (Message Passing Interface)

Наиболее распространённой технологией программирования параллельных компьютеров м распределённой памятью в настоящее время является MPI. Стандарт MPI фиксирует интерфейс, который должны соблюдать как система программирования MPI на каждой вычислительной системе, так и пользователь при создании свих программ. Современные реализации чаще всего соответствуют стандарту MPI версии 1.1. В 1997-1998 годах появился стандарт MPI 2.0, значительно расширяющий функциональность предыдущей версии. Однако, до сих пор этот стандарт не получил широкого распространения. MPI-программа – это множество параллельных взаимодействующих процессов. Все процессы порождаются один раз, образуя параллельную часть программы. В ходе выполнения MPI-программы порождение дополнительных процессов или уничтожение существующих не допускается. Каждый процесс работает в своём адресном пространстве, никаких общих переменных или данных в MPI нет. Основным способом взаимодействия между процессами является явная посылка сообщений. Для локализации взаимодействий параллельных процессов программы можно создавать группы процессов, предоставляя им общую среду для общения – коммуникатор. Состав образуемых групп произволен. Группы могут полностью входить одна в другую, не пересекаться или пересекаться частично. При старте программы всегда считается, что все порождённые процессы работают в рамках всеобъемлющего коммуникатора. Этот коммуникатор существует всегда и служит для взаимодействия всех процессов MPI-программы. Каждый процесс имеет уникальный атрибут номер процесса, который является целым неотрицательным

числом. С помощью этого атрибута происходит значительная часть взаимодействия процессов между собой.

1.2 Технология сбора данных

Первостепенную важность для оценки эффективности параллельной программы имеет информация о взаимодействиях между её отдельными частями (процессами) и долях времён, затраченных процессами на взаимодействие и ожидание. Для получения этой информации во время выполнения программы вызовы примитивов взаимодействия должны соответствующим образом обрабатываться средством трассировки. Рассмотрим основные подходы.

1.2.1 Синтаксический анализ исходных текстов

С помощью синтаксического анализатора в исходном тексте программы выявляются вызовы примитивов взаимодействия и заменяются вызовами профилирующей библиотеки. Основным преимуществом этого метода является возможность для каждого вызова передать трассировщику положение этого вызова в исходном тексте, имена переменных и другую информацию, не доступную во время выполнения. В результате чего, при анализе трассы программы можно будет легко сопоставить вызовы библиотеки передачи сообщений с исходным кодом. Также есть возможность легко вернуться к «рабочему» варианту программы, без трассирующих вызовов, т.е. не использовать разбор кода. Из недостатков следует выделить: сложность реализации, зависимость от языка программирования, дополнительный шаг компиляции, увеличение объёма трассы.

1.2.2 Статическое связывание с библиотекой профилировщика

В этом случае изменения исходного кода не требуются. Вместо этого на этапе связывания вызовы библиотеки передачи сообщений подменяются вызовами профилировщика. Конфликт имён при этом разрешается либо за счёт многоэтапного связывания, либо за счёт «слабых» (week) связей. Как и в предыдущем случае, здесь не требуется вносить изменения в исходный код, но при этом не требуется проводить сложный анализ исходного кода.

1.2.3 Определение точек трассировки во время выполнения

Метод основан на анализе частоты и шаблонов вызовов функций во время выполнения программы. При этом двоичный код программы компилируется стандартным образом, а профилирующие вызовы вставляются автоматически на основе реального поведения программы. Этот метод можно использовать при невозможности перекомпиляции программы. Но придётся столкнуться с такими проблемами как:

зависимость от конкретной платформы, отсутствие возможности прямо указать необходимость профилирования того или иного участка программы, сложность сопоставления обнаруженного узкого места с исходным кодом. Кроме получения информации о системных вызовах программы, также необходимо предоставить пользователю возможность указывать собственные временнЫе метки. Это позволяет определить время выполнения логических участков программы, а также упростить связь событий трассы с исходным кодом. Работа над выявлением узких мест в программе как правило ведётся итерационно, со всё большим уровнем детализации. При этом внимание разработчика сосредотачивается на отдельных участках программы, оставляя другие участки за кадром. Чтобы снизить объём трассы, средство трассировки должно допускать трассировку только части программы, иметь средства для приостановки и возобновления трассировки [3]. Изложенным требованиям удовлетворяет библиотека PICL, использующая для трассировки интерфейс профилирования MPI.

1.3 PICL

PICL (Portable Instrumented Communication Library) - библиотека подпрограмм, разработанная в Oak Ridge National Laboratory (ORNL), которая реализует общий нтерфейс передачи сообщений на множестве многопроцессорных архитектур. Программы, написанные с помощью примитивов PICL вместо команд среды взаимодействия, являются переносимыми в том смысле, что они могут выполняться на любой машине, для которой есть реализация PICL.
PICL была разработана для:

а) Обеспечения переносимости между разными платформами с минимальными накладными расходами;

б) Получения трассировочных данных по межпроцессным взаимодействиям и пользовательским событиям, также с минимальными накладными расходами для избежания влияния на поведение исследуемой задачи, и

в) Обеспечения переносимости на новые системы, простого расширения для использования преимуществ новых команд конкретной системы

Для достижения этих целей PICL реализована как библиотека подпрограмм языка C с машинно-независимым трассировочным слоем и машинно-зависимым слоем, отображающимся на команды конкретной системы (когда это возможно). PICL была открыта для широкого использования в марте 1989. Её разработка продолжалась, и были выпущены несколько релизов, добавляющие новые коммуникационные и трассировочные команды и поддержку новых вычислительных систем. В последней версии PICL запись трассы имеет формат, показанный в таблице 1.1. Для каждой записи обязательными являются первые шесть полей, но если количество полей данных равно нулю, то дескриптор данных исключается.

Таблица 1.1 — Формат записи трассы PICL

Наименование поля	Назначение
тип записи	тип информации в записи
тип события	тип события, с которым связана запись
отметка времени	когда информация была истинной
идентификатор процессора	процессор, с которым связана информация
идентификатор процесса	процесс, с которым связана информация
количество полей данных	количество дополнительных полей данных, связанных с данными типами записи и события
дескриптор данных	формат полей данных
данные	дополнительные поля данных

Основное внимание в этом формате уделено записи информации, связанной как с системными, так и с пользовательскими событиями, при этом каждая запись связана с определённым типом события. Порядок полей записи отражает важность полей для систем, обрабатывающих данные: что, где, когда и ассоциированные с событием данные. Если тип записи или события не знаком системе, то данную запись можно пропустить. Но если система хочет получить информацию из записи, то ей для этого всё предоставлено.

Формат PICL предусматривает ранение данных в текстовом виде. Все поля состоят из символов кодировки ASCII и разделены пробелами. Тип записи, тип события, идентификаторы процессора и процесса, количество полей данных – целые числа. Дескриптор данных – это либо целое число, либо символьная строка переменной длины, заключённая в двойные кавычки. Формат полей данных определяется дескриптором данных. Временная метка – число с плавающей точкой, разрешение которой зависит от единиц измерения (например, секунд) и точности таймера на машине, на которой генерируется трасса.

Практика показала, что текстовый формат очень полезен для переносимости, поиска и исправления ошибок в трассе и для возможной работы с трассой без помощи средств визуализации, и он не препятствует сжатию файлов, если это необходимо. При этом PICL использует двоичное представление данных при внутренней обработке для экономии места и избежания накладных расходов при преобразовании данных [4]. Значения полей и их интерпретация приведены в таблице 1.2 Идентификаторы процесса и процессора определяют «местоположение» события. Хотя обычно процессы на каждом процессоре нумеруются последовательно начиная с 0, может быть полезно нумеровать процессы независимо от процессора, прямо поддерживая как процессоро-зависимый, так и процессоро-независимый подход к визуализации. (Заметим, что на данный момент PICL не поддерживает много процессов на одном

Таблица 1.2 — Значения полей записи

Поле	Назначение
тип записи	$\geqslant 0$ пользовательский тип записи ≤ 0 стандартный системный тип записи
тип события	$\geqslant 0$ пользовательское событие или определённое пользователем подмножество событий $= -1$ относится ко всем событиям (глобальная информация) < -1 системное событие или системное подмножество событий
временная метка	$\geqslant 0$идентификатор конкретного процессора $= -1$ относится ко всем процессорам (глобальная информация) < -1 относится к подмножеству процессоров
идентификатор процесса	$\geqslant 0$ идентификатор процесса на заданном процессоре $= -1$ относится ко всем процессам на заданном процессоре (глобальная информация) < -1 относится к подмножеству процессов на указанном процессоре
количество полей данных	количество дополнительных полей данных
дескриптор данных	если заключён в двойные кавычки, то формат функции scanf(), иначе: $= 0$ символ ("%c") $= 1$ строка ("%s") $= 2$ целое число ("%d") $= 3$ длинное целое число ("%ld") $= 4$ число с плавающей точкой одинарной точности ("%f") $= 5$ число с плавающей точкой двойной точности ("%lf") > 6 другой предопределённый формат данных

(последовательном) процессоре. Поле идентификатора процесса было введено для совместимости с системами, которые поддерживают эту модель и для возможности будущего расширения PICL. В трассах текущего формата идентификатору процесса присваивается либо действительной PID, т.е. номер процесса, заданный операционной системой, либо -1.

2 Конструкторский раздел

2.1 Общая архитектура приложения

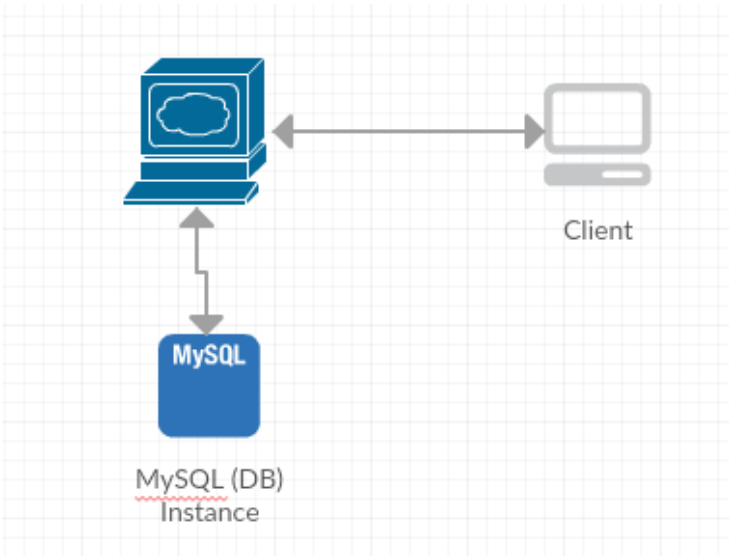

Рисунок 2.1 — Общая архитектура приложения

Приложение построено по архитектуре "клиент-сервер". Клиент запускает на своём компьютере приложение и взаимодействует посредством сети с удаленным сервером. Сервер позволяет осуществлять доступ к необходимой информации из базы данных, разграничивать доступ пользователей к той или иной информации. Так же на сервере находится приложение, позволяющее парсить trace-файлы программы и заносить информацию в базу данных.)

2.2 Клиентская часть

Клиентское приложение позволяет осуществлять авторизацию, получать информацию о доступном ему списке файлов трасс, получать общую информацию о файле трасс, получать детальную информацию о файле трасс в заданном временом временном интервале с заданным количеством процессов.

MainWindow. Класс графического интерфейса приложение. Реализует логику взаимодействия между клиентом и программой.

NetworkManager. Класс работы с сетью. Реализует логику асинхронных запросов к серверу.

RequestManager. Класс работы с телами ответов сервера.Парсит приходяющую с сервера информацию.

TableManager. Класс работы с визуалиацией приходящей информации. Реализует логику работы с таблицей отображения trace-информации.

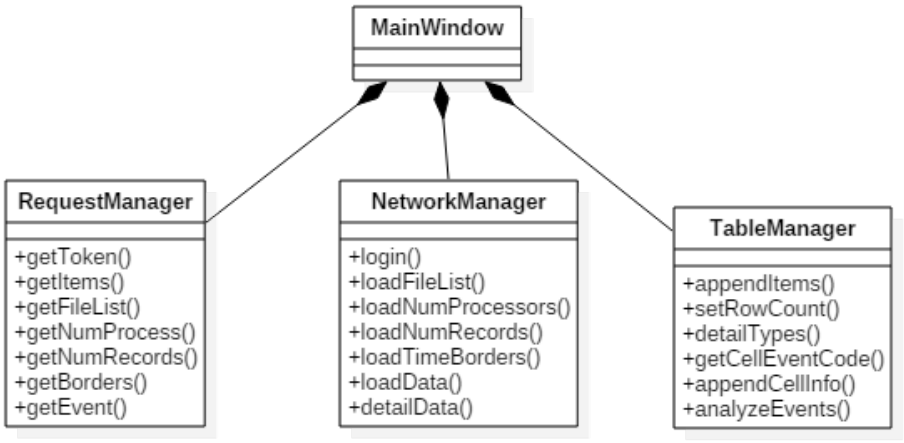

Рисунок 2.2 — Диаграмма классов клиентского приложения

2.3 Серверная часть

Сервер был разработан на базе NodeJS express. Он позволяет асинхронно работать с поступающим ему запросами. Код сервера представляет собой множество функций-обработчиков приходящих запросов.

```
1 apiRoutes.get('/authenticate', function (req, res) {..}
2 apiRoutes.get("/getFile", function (req, res) {...}
3 apiRoutes.get("/getTimeBorders", function (req, res) {...}
4 apiRoutes.get("/getFileList", function (req, res) {...}
5 apiRoutes.get("/getNumRecords", function (req, res) {...}
6 apiRoutes.get("/getCodeInfo", function (req, res) {...}
7 apiRoutes.get("/getNumProcesses", function (req, res) {...}
```

2.3.1 API для взаимодействия с сервером

Была разработано API взаимодействия между клиентом и сервером. Безопастность взаимодействия гарантируется тем, что взаимодействия осуществляется на основе токенов. Все запросы, кроме запроса на авторизацию, требуют содержания в себе уникального токена.Токен дается клиенту после успешного прохождения авторизации.

```
1  [GET] /authenticate
2  Авторизация
3  Аргументы
4  )name – логин
5  )password – пароль
6
7  [Example response]
8  {
9  "success": true,
10 "message": "Enjoy your token!",
```

```
"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIj
oiVXNlciIsInBhc3N3b3JkIjoicHdkI
iwiaWF0IjoxNDk2NDM4ODc4LCJleHAiOjE0OT
Y1MjUyNzh9.0bhg6Ct8YXOGixVY-ZukmdVj
-35AE0BSZExGWJlpNKQ"
}
```

[GET] /getFile

Получение куска содержанимого файла

Аргументы

)filename – имя файла

)offset – отступ от начала файла

)limit – количество получаемых записей

)timeMin – время начала записей

)timeMax – время конца записей

[Example response]

```
{
"result": [
{
"filename": "trace1",
"typeRecord": -3,
"typeEvent": -901,
"time": 0.000007,
"prid": 0,
"pid": 0,
"numData": 0,
"data": "-1"
}],
}
```

[GET] /getTimeBorders

Получение временного интервала трассы

Аргументы

)filename – имя файла

[Example response]

```
{
"result": [
{
"MIN(time)": 0.000007000000096013537,
"MAX(time)": 0.11629900336265564
}],
}
```

[GET] /getFileList

Получение списка файлов, доступных для данного пользователя

```
Аргументы
)name – логин

[Example response]
{
"success": true,
"message": [
{
"filename": "trace1"
},
{
"filename": "trace2"
}],
}

[GET] /getNumRecords
Получение количества записей для данного файла
Аргументы
)filename – имя файла
)timeMin – время начала записей
)timeMax – время конца записей

 [Example response]
 {
 "result": [
 {
 "COUNT": 522
 }],
 }

[GET] /getCodeInfo
Получение информации о событие по его коду
Аргументы
)code – код события

[Example response]
{
"result": [
{
"name": "WILDCARD",
"description": "wildcard for all or any processes",
"category": "event type definitions"
}],
}

[GET] /getNumProcess
Получение максимального количества процессов, которые записаны в файл
```

```
105 Аргументы
106 )filename – имя файла
107
108 [Example response]
109 {
110 "result": [
111 {
112 "max(prid)": 7
113 }],
114 }
```

2.4 Взаимодействие клиента и сервера

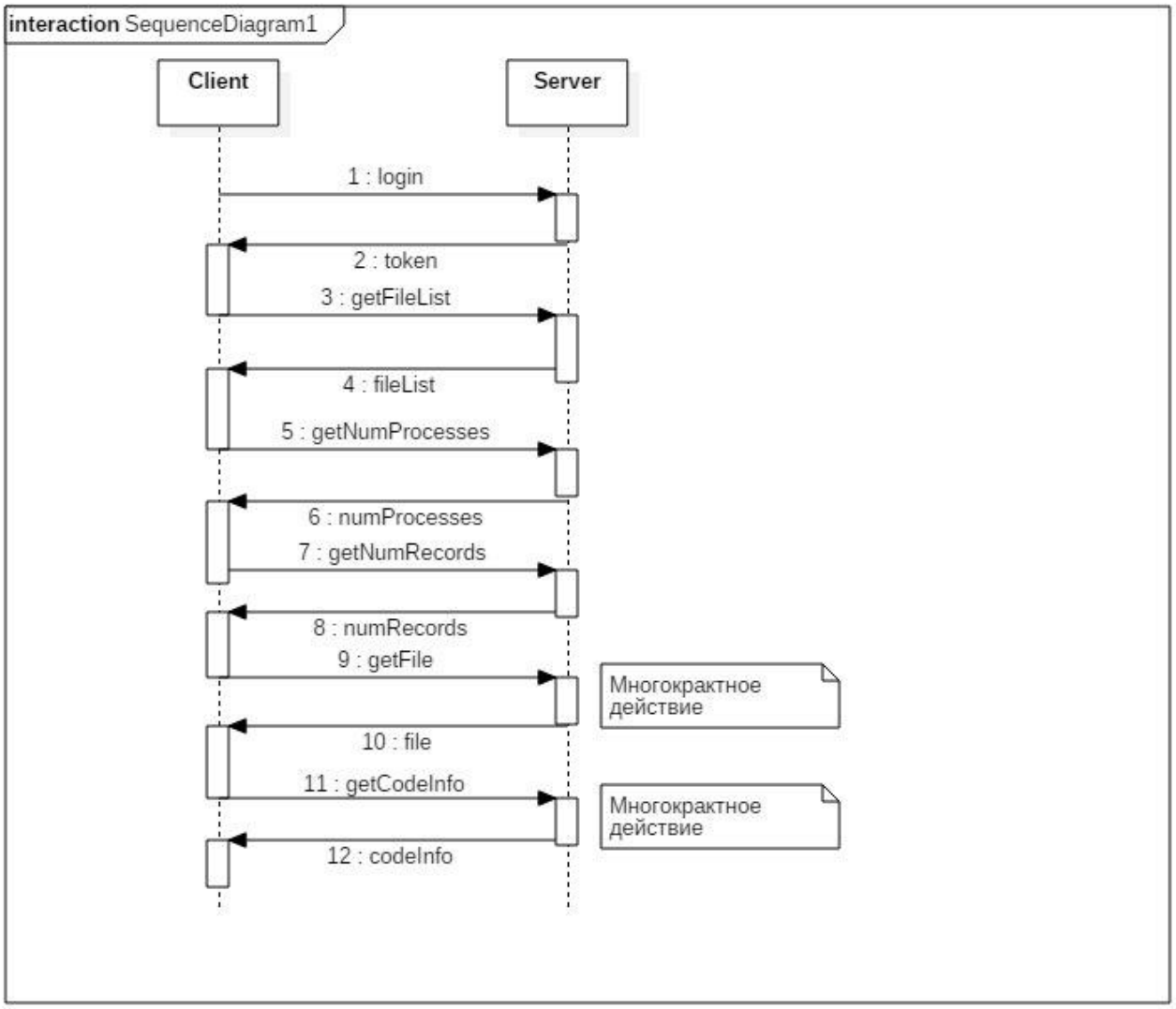

Рисунок 2.3 — Диаграмма взамодействия клиента и сервера

В начале работы приложения клиент должен пройти авторизацию(1). В случае успешного прохождения авторизации ему посылается token,позволяющий ему

получить права доступа к работе с trace-информацией(2). После этого клиент загружает список доступных ему файлов(3-4). Запросы (5-8) технические и нужны для корректного отображения trace-информации на стороне клиента. После того как вся техническая информация была загружена, клиент приступает к загрузке файла(9-10). Файл загружается «небольшими порциями», поэтому возможно многократное повторение запросов (9-10) до успешной загрузки файла. В случае успешной загрузки файла клиент может получить детальную информацию об этом файле при помоще запросов (11-12).

2.5 Синтаксический анализ trace-файлов

Для синтаксического анализа(парсинга) trace-файлов и занесения их в базу данных была разработана консольная программа.

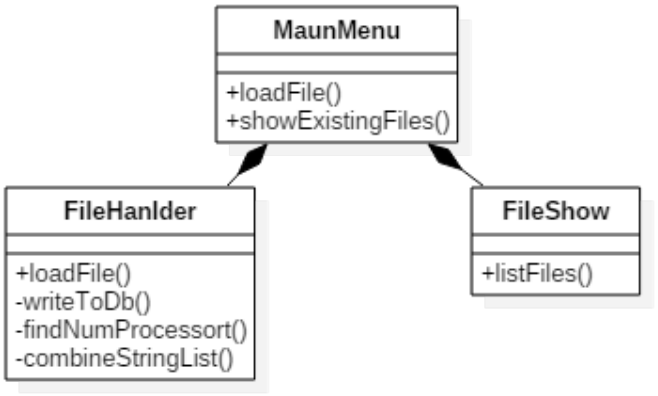

Рисунок 2.4 — Диаграмма классов серверного приложения

MainMenu. Класс главного меню программы. Реализует логику взаимодействия между клиентом и программов.
FileShow. Класс для работы с файловой системой компьютера.
FileHandler. Класс для парсинга trace-файла. Реализует логику анализа trace-файла и занесения его в базу данных.

2.6 Базы данных

Для хранения файлов трасс формата PICL была спроектирована база данных.

```
1 CREATE TABLE Tracks (
2 filename VARCHAR(50) NOT NULL,
3 typeRecord INT NOT NULL,
4 typeEvent INT NOT NULL,
5 time FLOAT NOT NULL,
6 prid INT NOT NULL,
```

```
7  pid INT NOT NULL,
8  numData INT NOT NULL,
9  data VARCHAR(200),
10 FOREIGN KEY (typeEvent) REFERENCES Codes(code)
11 ON DELETE CASCADE ON UPDATE CASCADE,
12 FOREIGN KEY (filename) REFERENCES Files(filename)
13 ON DELETE CASCADE ON UPDATE CASCADE
14 ) ENGINE = INNODB DEFAULT CHARSET=utf8;
```

Так же были спроектированы вспомогательные базы данных для хранения информации о событиях MPI, файлах, пользователях.

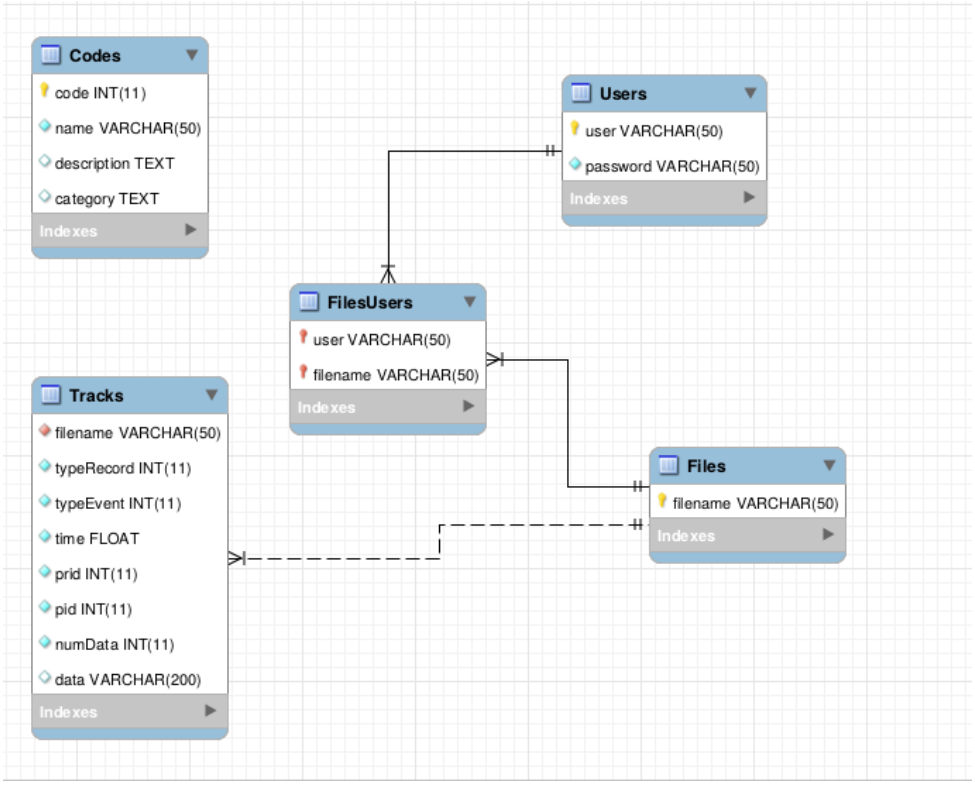

Рисунок 2.5 — Диаграмма классов серверного приложения

```
1  CREATE TABLE Users(
2  user VARCHAR(50) NOT NULL PRIMARY KEY,
3  password VARCHAR(50) NOT NULL
4  ) ENGINE = INNODB DEFAULT CHARSET=utf8;
5
6  CREATE TABLE Files(
7  filename VARCHAR(50) NOT NULL PRIMARY KEY
8  ) ENGINE = INNODB DEFAULT CHARSET=utf8;
9
10 CREATE TABLE FilesUsers(
11 user VARCHAR(50) NOT NULL,
12 filename VARCHAR(50) NOT NULL,
13 PRIMARY KEY (user, filename),
```

```
14 FOREIGN KEY (filename) REFERENCES Files(filename)
15 ON DELETE CASCADE ON UPDATE CASCADE
16 ) ENGINE = INNODB DEFAULT CHARSET=utf8;
17
18 CREATE TABLE Codes(
19 code INT NOT NULL PRIMARY KEY,
20 name VARCHAR(50) NOT NULL,
21 description TEXT,
22 category TEXT
23 ) ENGINE = INNODB DEFAULT CHARSET=utf8;
```

2.6.1 Нормальные формы

Нормальная форма — свойство отношения в реляционной модели данных, характеризующее его с точки зрения избыточности, потенциально приводящей к логически ошибочным результатам выборки или изменения данных. Нормальная форма определяется как совокупность требований, которым должно удовлетворять отношение. Процесс преобразования отношений базы данных (БД) к виду, отвечающему нормальным формам, называется нормализацией. Нормализация предназначена для приведения структуры БД к виду, обеспечивающему минимальную логическую избыточность, и не имеет целью уменьшение или увеличение производительности работы или же уменьшение или увеличение физического объёма базы данных. Конечной целью нормализации является уменьшение потенциальной противоречивости хранимой в базе данных информации. Как отмечает К. Дейт, общее назначение процесса нормализации заключается в следующем:

а) исключение некоторых типов избыточности;

б) устранение некоторых аномалий обновления;

в) разработка проекта базы данных, который является достаточно «качественным» представлением реального мира, интуитивно понятен и может служить хорошей основой для последующего расширения;

г) упрощение процедуры применения необходимых ограничений целостности.

Устранение избыточности производится, как правило, за счёт декомпозиции отношений таким образом, чтобы в каждом отношении хранились только первичные факты (то есть факты, не выводимые из других хранимых фактов)[5].

Первая нормальная форма. Переменная отношения находится в первой нормальной форме тогда и только тогда, когда в любом допустимом значении отношения каждый его кортеж содержит только одно значение для каждого из атрибутов. В реляционной модели отношение всегда находится в первой нормальной форме по определению понятия отношение. Что же касается различных таблиц, то они могут не быть правильными представлениями отношений и, соответственно, могут не нахо-

диться в 1NF. В соответствии с определением К. Дж. Дейта для такого случая, таблица нормализована (эквивалентно — находится в первой нормальной форме) тогда и только тогда, когда она является прямым и верным представлением некоторого отношения. Конкретнее, рассматриваемая таблица должна удовлетворять следующим пяти условиям:

а) Нет упорядочивания строк сверху-вниз (другими словами, порядок строк не несет в себе никакой информации).

б) Нет упорядочивания столбцов слева-направо (другими словами, порядок столбцов не несет в себе никакой информации).

в) Нет повторяющихся строк.

г) Каждое пересечение строки и столбца содержит ровно одно значение из соответствующего домена.

д) Все столбцы являются обычными (в таблице нет «скрытых» компонентов, которые могут быть доступны только в вызове некоторого специального оператора взамен ссылок на имена регулярных столбцов).

В моей базе данных каждый кортеж всех таблиц содержит только атомарные атрибуты (атрибут атомарен, если его значение теряет смысл при любом разбиении на части или переупорядочивании), отсутствуют повторяющиеся кортежи. Исходя из этого, можно сказать, что все таблицы находятся в первой нормальной форме.

Вторая нормальная форма. Переменная отношения находится во второй нормальной форме тогда и только тогда, когда она находится в первой нормальной форме и каждый неключевой атрибут неприводимо зависит от её потенциального ключа. Неприводимость означает, что в составе потенциального ключа отсутствует меньшее подмножество атрибутов, от которого можно также вывести данную функциональную зависимость. Для неприводимой функциональной зависимости часто используется эквивалентное понятие «полная функциональная зависимость». Если потенциальный ключ является простым, то есть состоит из единственного атрибута, то любая функциональная зависимость от него является неприводимой (полной). Если потенциальный ключ является составным, то согласно определению второй нормальной формы в отношении не должно быть неключевых атрибутов, зависящих от части составного потенциального ключа. Вторая нормальная форма по определению запрещает наличие неключевых атрибутов, которые вообще не зависят от потенциального ключа. Таким образом, вторая нормальная форма запрещает создавать отношения как несвязанные (хаотические, случайные) наборы атрибутов. В моей базе данных не все потенциальные ключи состоят из одного атрибута, следовательно, нельзя сказать, что все таблицы находятся во второй нормальной форме.

3 Технологический раздел

3.1 Выбор языка программирования

Для реализации был выбран язык C++. Данный язык был обоснован следующими причинами: Причины:

а) Его поддерживает библиотека PICL

б) Компилируемый язык со статической типизацией.

в) Сочетание высокоуровневых и низкоуровневых средств.

г) Реализация ООП.

д) Наличие удобной стандартной библиотеки шаблонов

3.2 Выбор вспомогательных библиотек

Для реализации программы была выбрана библиотека Qt.

а) Широкие возможности работы с изображениями, в том числе и попиксельно

б) Наличии более функциональных аналогов стандартной библиотеки шаблонов в том числе для разнообразных структур данных

Так же были были использованы библиотеки Nodejs,express,jsonwebtoken,mysql-nodejs

3.3 Выбор базы данных

Для хранения файлов трасс была выбрана база данных MySQL

а) Быстродействие. Благодаря внутреннему механизму многопоточности быстродействие MySQL весьма высоко.

б) Лицензия. Раньше лицензирование MySQL было немного запутанным; сейчас эта программа для некоммерческих целей распространяется бесплатно

в) Переносимость. В настоящее время существуют версии программы для большинства распространенных компьютерных платформ. Это говорит о том, что вам не навязывают определенную операционную систему. Вы сами можете выбрать, с чем работать, например с Linux или Windows, но даже в случае замены ОС вы не потеряете свои данные и вам даже не понадобятся дополнительные инструменты для их переноса.

Заключение

В данной работе было реализовано клиент-серверное приложение для анализа работы параллельных программ. Приложение позволяет получить информацию о том когда происходило какое-либо событие в программе и информацию об этом событии. Программа не привязана к какой-то конкретной операционной системе и может быть скомпилирована и запущена на всех популярных ОС.

Список использованных источников

1. *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления / Воеводин Вл.В. Воеводин В.В. — БХВ-Петербург, 2002.

2. *O.N, D'Paola.* Performance visualization of parallel programs / D'Paola O.N. — University of Southampton, 1997.

3. *Haake B. Schauser K.E., Scheiman C.J.* Profiling a parallel language based on fine-grained communication / Scheiman C.J. Haake B., Schauser K.E. — University of California, Santa Barbara, 2001.

4. Portable Instrumentation Library. — `http://www.csm.ornl.gov/picl/index.html`.

5. *Дейт., К. Дж.* Введение в системы баз данных / К. Дж. Дейт. — М.: «Вильямс», 2006.

6. RESTful API на Node.js + MongoDB. — `https://habrahabr.ru/post/193458/`.