МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра Параллельных вычислительных технологий

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Дубовика Антона Сергеевича
(фамилия, имя, отчество автора)

Реализация системы визуализации исполнения фрагментированных программ

Направление подготовки **010500 - «Прикладная математика и информатика»**

Руководитель	**Автор**
Малышкин В.Э.	**Дубовик А.С.**
(фамилия, И.О.)	(фамилия, И.О.)
д.т.н, профессор	**ФПМИ, ПМ-63**
(уч. степень, уч. звание)	(факультет, группа)
(подпись, дата)	(подпись, дата)

Новосибирск, 2010 г.

Содержание

Введение

Существуют научные задачи, решение которых требует больших вычислительных ресурсов – больших, чем может предоставить один компьютер. Такие трудоёмкие задачи как экономическое прогнозирование, сейсмоанализ, разработка и изучение свойств новых лекарств, моделирование задач вычислительной аэро- и гидродинамики требуют терабайты памяти и годы процессорного времени при последовательном исполнении. В подобных случаях используются параллельные вычисления.

В общем случае параллельные вычисления – это использование множества вычислителей для решения одной задачи. Программа разбивается на множество частей, которые могут быть исполнены независимо друг от друга, каждая часть является набором инструкций, исполняющихся последовательно. Таким образом, инструкции от каждой части могут исполняться на разных вычислителях одновременно.

Реальный мир имеет параллельную природу – множество сложных, взаимодействующих событий происходят одновременно, например, процессы образования галактик, движения планет, погодные явления, смещения тектонических плит и т.д. Поэтому использование параллельных вычислений востребовано в решении сложных научных и технических проблем, имеющих начало в реальном мире.

Современные промышленные приложения оказывают не меньшее влияние на развитие параллельных вычислений. Такого рода приложения требуют изощренной обработки сотен гигабайт данных, например, задачи из области Data Mining, криптографии, биоинформатики и экономики.

Основными аргументами [1] в пользу параллелизма являются:

1. **Увеличение количества операций с единицу времени**. Одним из эмпирических наблюдений, подобно закону Мура, является тот факт, что каждые 18 месяцев количество вычислительных мощностей, доступных за данную цену, удваивается. То, каким образом нужно использовать транзисторы, чтобы увеличить количество вычислений в единицу времени, является сложной технологической и архитектурной задачей. Одним из решений данной проблемы является использование параллелизма – многоядерность процессора, суперскалярность, конвейеризация выполнения инструкций.
2. **Большая пропускная способность и объем памяти**. Общая скорость вычислений определяется не только скоростью CPU, но и тем, насколько быстро память может предоставить процессору запрашиваемые данные. Темпы технологического прогресса таковы, что частота процессоров увеличивается большими темпами, чем время доступа в оперативную память. Это растущее несоответствие между скоростью процессора и латентностью памяти решается иерархией типов памяти с быстрым доступом – кэш-памятью. Общая производительность памяти определяется пропускной способность и объемом памяти. В параллельных компьютерах производительность памяти улучшается за счет суммарно большего объема памяти и большей общей пропускной способности памяти. Кроме того, возможен эффект суперлинейного ускорения, когда время работы на n вычислителях более чем в n раз меньше времени исполнения программы на одном. Такое возможно из-за того, что большее количество данных программы помещается в кэш-память, и, следовательно, большее количество запросов в память удовлетворяется из кэш-памяти – самой быстрой памяти в иерархии видов памяти компьютера.
3. **Нежелательность централизованного подхода при решении определенного класса задач**. Существуют задачи, которые естественно решать распределено. Например, обработка метеоданных, собираемых с множества станций, распределенных по территории страны. В таких случаях централизованный подход, когда данные с множества компьютеров обрабатываются на одном – центральном компьютере, неосуществим или неэффективен.

Применение параллельных компьютеров

Параллельные компьютеры нашли множество применений от моделирования реальных физических процессов в научных и инженерных приложениях до Data mining и обработки транзакций в коммерческих приложениях.

Промышленное проектирование

Параллельные компьютеры в промышленном проектировании применяются для решения дискретных и непрерывных оптимизационных задач. Метод ветвей и границ для задач линейной оптимизации, генетические алгоритмы для дискретной оптимизации эффективно распараллеливаются и часто используются при проектировании формы крыльев, двигателей внутреннего сгорания, высокоскоростных электрических цепей.

Научные проблемы

Анализ органических соединений с целью разработки новых лекарств и медицинских препаратов требует использование эффективных алгоритмов и огромных вычислительных ресурсов.

Результаты исследований в области вычислительной физики и химии, направленных на понимание процессов, происходящих в масштабах от квантовых эффектов до макромолекулярных структур, помогают проектировать новые материалы с необходимыми физическими свойствами. Параллельные компьютеры применяются в астрофизике в исследовании эволюции галактик, термоядерных процессов внутри звезд и при обработке чрезвычайно больших объемов данных, поступающих от телескопов. Моделирование погоды, разведка полезных ископаемых, предсказание наводнений – при решении этих важных для повседневной жизни людей задач используются параллельные компьютеры.

Биоинформатика и астрофизика представляют наиболее сложные проблемы с точки зрения анализа больших объемов данных, таких как, базы данных по экспрессии генов, библиотеки геномов, последовательностей РНК, беков и белковых мотивов, подробнейшие карты звездного неба. Эффективный анализ этих данных содержит в себе ключ к значительным научным открытиям и требует гигантских вычислительных мощностей.

Коммерческое применение

Параллельные платформы от мультипроцессоров до кластеров используются в качестве web-серверов и серверов баз данных. Например, супервычислительной сети Нью-Йоркской фондовой бирже на Уолл-стрит ежедневно необходимо обрабатывать миллионы запросов от сотен тысяч пользователей. Ввиду доступности информации о транзакциях, имеет смысл применять методы Data mining для принятия эффективных маркетинговых решений. Большой объем и распределенный характер такого рода данных требует использование эффективных параллельных алгоритмов для таких задач как поиск ассоциативных правил, классификация, кластеризация и анализ временных рядов.

Разработка параллельных программ

Процесс разработки параллельных программ (ПП) существенно более сложный, чем последовательных.

На этапе проектирования ПП необходимо разработать эффективный параллельный алгоритм (зачастую на основе существующего последовательного алгоритма), то есть определить способ распределения и обработки данных на отдельных вычислителях и характер их взаимодействия, направленного на удовлетворение информационных зависимостей. Необходимо также доказать полную корректность параллельного алгоритма.

При программной реализации параллельного алгоритма важно заложить в программу определенные динамические свойства и запрограммировать межпроцессорные коммуникации, для чего необходимы знания архитектуры ЭВМ и теории параллельного программирования.

Параллельные вычисления целесообразно использовать, когда невозможно последовательно решить задачу из-за недостатка памяти или неприемлемого времени вычислений.

Таким образом, необходима такая организация вычислений в ПП, при которой вычислительные ресурсы мультикомпьютера используются наиболее полно, что, фактически, означает минимизацию времени или ресурсов исполнения программы. Необходимым условием для эффективного использования ресурсов мультикомпьютера является обеспечение следующих динамических свойств ПП [2].

1. **Настройка на доступные ресурсы мультикомпьютера** – настройка ПП, учитывающая структуру и характеристики мультикомпьютера (количество вычислительных узлов, отношение коммуникационных связей между отдельными узлами, пропускная способность коммуникационных связей, архитектурные особенности вычислительных узлов – количество ядер/процессоров, размер оперативной и кэш-памяти, характеристики CPU) и позволяющая до начала вычислительного процесса равномерно распределить нагрузку между вычислительными узлами.
2. **Динамическая балансировка загрузки** – распределение вычислительной загрузки между процессами в ходе исполнения ПП, таким образом, чтобы все процессы были заняты все время, что эквивалентно минимизации времени простоя процесса.
3. **Переносимость в классе параллельных компьютеров**. ПП должна хорошо исполняться на каждом параллельном компьютере из некоторого класса, например, на кластере с топологией двумерная решетка.
4. **Динамический учет свойств задачи**. Параллельная программа должна учитывать особенности вычислительных моделей, например, характер изменения потребляемых вычислительных ресурсов во времени.
5. **Реализация межпроцессорных коммуникаций на фоне вычислений** – параллельная программа должна использовать свойство компьютера, обеспечивающее независимость работы центрального процессора от сетевой карты.

Процессы верификации, отладки и сопровождения параллельных программ сложны для человека. Это обусловлено тем, что порядок выполнения процессов в ПП жестко не фиксирован, что является причиной недетерминизма исполнения. При каждом выполнении программы ее процессы исполняются, вообще говоря, в другом порядке, что вызывает трудности при локализации ошибок.

Таким образом, при разработке ПП программисту приходится уделять внимание системным проблемам параллельного программирования. Поэтому актуальна разработка такой системы программирования, которая возьмет на себя технические вопросы конструирования ПП и позволит программисту сосредоточится на алгоритме решения задачи.

Визуализация данных программы

Визуализация данных – это распространённый подход, помогающий человеку понимать сложные явления и анализировать большие объемы данных. Зачастую, поведение ПП крайне сложно и проследить взаимодействие и порядок исполнения множества процессов, анализируя мегабайты лог-файлов, не представляется возможным. В таких случаях естественно использовать визуализацию для лучшего понимания хода выполнения ПП, выявления ее узких мест и, как следствие, оптимизации.

Рассмотрим ряд программных продуктов, использующих визуализацию, для облегчения процесса разработки параллельных программ.

Обзор средств визуализации исполнения параллельных программ

TotalView [3] – наиболее популярный отладчик в сфере высокопроизводительных вычислении (используется на 98 из 100 суперкомпьютеров из top100 [4]). Применяется для отладки и анализа на предмет утечек памяти как последовательных, так и параллельных программ, использующих OpenMP, MPI, POSIX threads. Поддерживает программы, написанные на C/C++, Fortran 77/90, Assembler.

TotalView обеспечивает графическое отображение графов вызова функций и визуализацию массивов данных в виде трехмерных и двумерных графиков.

Существует только Linux-реализация.

Jumpshot [5] – кроссплатформенное средство визуализации для анализа производительности параллельных программ, распространяется вместе с библиотекой расширений MPE, предоставляемой большинством реализаций MPI .

Jumpshot для каждого MPI-процесса отображает интервал времени в течение, которого он исполнялся с обозначением промежутков времени, затраченных на исполнение или ожидание MPI команд (MPI_Send, MPI_Recv,MPI_Reduce и т.д.), стрелками обозначается передача сообщения между процессами.

Информацию об исполнении программы Jumpshot считывает из лог-файлов, записанных MPI-программой, т.е. визуализация возможна только после исполнения параллельной программы.

ParaGraph [6] – средство визуализации хода исполнения MPI-программ, подобно Jumpshot использующее информацию, собранную в результате исполнения MPI программы. Функции сбора трассировочной информации осуществляет профилирующая библиотека MPICL. ParaGraph обеспечивает анимационное представление исполнения программы.

ParaGraph обеспечивает несколько различных способов отображения загруженности процессоров (диаграммы Ганта, Кивиата и т.д.) и коммуникационного трафика (графики количества переданных данных, пространственнно-временные диаграммы с отображением моментов передачи и приема сообщений, отображение длины очередей сообщений, коммуникационная матрица).

ParaGraph считывает записанный параллельной программой лог-файл и проигрывает во времени отображение представления параллельной программы в соответствии с временными отметками событий.

Существует только Linux-реализация.

The Rivet Visualization Environment [7] —система визуализации для изучения сложных вычислительных систем, проект Стэнфордского университета. Данная система позволяет генерировать конкретную визуализацию под исследуемую вычислительную систему.

Пользователь системы Rivet:

1. задает источник данных, как правило, лог-файлы;
2. определяет сеть трансформаций (*transformation network*), обрабатывающую данные из источника и выдающую в итоге таблицы данных (*data tables*), которые необходимо визуализировать;
3. задает визуальное представление таблиц данных, например, график или гистограмма.

The Projections Performance Analysis Framework [8] – фреймворк для анализа и производительности для программ, написанных на языке параллельного программирования Charm++. Данный фреймворк включает визуализатор, использующий лог-файлы, записанные ПП, для отображения:

1. диаграмм загруженности процессорных элементов;
2. временной шкалы, показывающей чем занимался конкретный процессорный элемент (ПЭ) в определенный момент времени;
3. статистики ПЭ, связанная с межкоммуникационным взаимодействием: общее число принятых/отосланных процессорным элементом сообщений, количество принятых/отосланных байт и пр.

Intel Thread Profiler [9] предназначен для профилирования многопоточных приложений, использующих OpenMP, Windows threads, POSIX threads, и помощи разработчику в процессе оптимизации ПП.

ITP осуществляет сбор и отображение после завершения ПП следующей информации:

1. затраты на синхронизацию, накладные расходы на поддержку работы с потоками;
2. дисбаланс вычислительной нагрузки между потоками;
3. сравнение результатов различных запусков;
4. критический путь программы;
5. распределение временных затрат по критическому пути на исполнение, синхронизацию, ожидание и блокировку;
6. связь потоковых событий с исходным кодом.

Отсутствует поддержка MPI.

Результаты анализа

Анализ показал, большинство средств визуализации исполнения ПП осуществляют визуализацию постфактум (post-mortem visualization [10]) – уже после завершения ПП, используя записанные ей лог-файлы (trace files), в которых записана последовательность событий ПП с временными отметками (time stamps) – межпроцессорные взаимодействия, изменение активности процесса, доступ на чтение/запись к объекту памяти, события, определенные программистом и т.д. В случае использования post-mortem систем анализ ПП на основе визуализации ее исполнения возможен только после завершения программы, что является существенным недостатком в использовании подобных СВ, когда вычисления выполняются в течение продолжительного времени, и важно своевременно заметить нежелательный ход исполнения программы, например, неэффективное распределение ресурсов или рост время простоя процессов, вызванный несинхронностью системных часов на узлах кластера.

Существует другой подход – визуализация в процессе исполнения ПП (real-time visualization), заключающийся в том, что состояние отображается одновременно с ходом исполнения ПП. Такой подход может быть реализован, например, с помощью централизованной схемы – параллельная программа отсылает сообщение о событии непосредственно после его происхождения системе визуализации, которая исполняется на одном вычислителе.

Разработка real-time системы визуализации сопряжена с рядом проблем, отсутствующих в post-mortem подходе, что объясняет распространенность последнего.

Недостатки real-time визуализации заключаются в следующем:

1) необходимо выделить один или более вычислителей, которые берут на себя функции обработки данных, поступающей от вычислителей, занятых исполнением ПП, и непосредственно визуализации. Накладные ресурсы на дополнительные коммуникационные

взаимодействия между вычислителями, обеспечивающие передачу трассировочной информации, могут негативно сказаться на производительности параллельной программы;

2) существует ограничение сверху на общее число вычислителей при использовании такого рода систем визуализации (СВ) – при большом количестве вычислителей СВ не успеет своевременно обрабатывать приходящую информацию, а, следовательно, отображать данные в режиме реального времени;

3) возможна ситуация, когда сообщение о двух событиях, происшедших в ПП в одном порядке, приходит к СВ в другом. Таким образом, система может прийти к неверному представлению ПП. Такую ситуацию надо каким-то образом обрабатывать. Выделим несколько проектных решений, комбинация которых может использоваться в централизованной схеме СВ для преодоления данной проблемы:

1. синхронная посылка узлу-визуализатору накопленной за определенный интервал времени трассировочной информации. Таким образом, СВ будет известен полный список событий, произошедших до определенного момента, а, следовательно, можно точно отобразить представление ПП. Недостаток такого решения заключается в накладных расходах на дополнительную синхронизацию и «прерывистой» визуализации исполнения ПП.
2. отслеживание моментов, когда данные приходят не в хронологическом порядке и приведение представления ПП к такому виду, как если бы порядок всех пришедших данных был верным. То есть, если очередное событие *A*, о котором пришла информация, произошло раньше, чем последнее обработанное СВ событие *B* (данное утверждение проверяется сравнением временных отметок событий), то происходит откат СВ к ряду событий, более ранних, чем *A*, осуществляется учет события *A* и последовательности событий, ранее отмененных, произошедших позже *A*. Заметим, что учет события *A* может нарушать некоторый инвариант СВ, например, условие положительности объема памяти, выделенного на узле. Однако, в конце работы ПП система визуализации будет обладать полным списком событий в верном порядке, поэтому представление ПП будет полным и корректным. Далее данный подход будет называться подходом сохранения порядка.
3. допущение неточного представления ПП и нарушения инвариантов – все события учитываются в том порядке, в котором приходят в СВ. Наиболее простое решение, которое, тем не менее, часто приемлемо.

4) части параллельной программы могут генерировать суммарно настолько большой объем трассировочных данных, что система визуализации будет не успевать обрабатывать ее и отображать. Проблема можно решить использованием децентрализованной двухуровневой схемы – разбиение множества вычислительных узлов на группы, каждой из которых назначается отдельный узел, который обрабатывает данные от этой группы и отсылает узлу-визуализатору.

5) важным требованием к СВ исполнения ПП, основанных на обработке трассировочной информации, является синхронизованность системных часов на узлах мультикомпьютера и высокая точность при генерации временных отметок событий ПП. Именно по временным отметкам система визуализации восстановит хронологический порядок событий, который в силу свойства недетерменизма исполнения ПП, особенно важен при анализе программы.

Фрагментированное программирование

Визуализация исполнения параллельной программы важна, как средство, позволяющее эффективно бороться со сложностью процесса разработки ПП, обусловленной вышеизложенными проблемами параллельного программирования как такового. Существует

широкий спектр программных продуктов и технологий разработки ПП, где востребована визуализация. Одна их таких технологий – технология фрагментированного программирования (ТФП) [11], разрабатываемая в ИВМиМГ СО РАН.

Основной задачей ТФП является автоматическое обеспечение динамических свойств параллельной программы, таких как, настройка на имеющиеся аппаратные ресурсы мультикомпьютера, динамическая балансировка нагрузки, осуществление коммуникационных взаимодействий на фоне вычислений.

Идея фрагментированного программирования заключается в представлении ПП в виде множества фрагментов вычислений, фрагментов данных и отношения частичного порядка на множестве фрагментов вычислений, ограничивающего порядок исполнения фрагментов вычислений. Фрагмент вычислений может быть грубо представлен как процедура с набором входных и выходных фрагментов данных. Представление алгоритма в ТФП называется фрагментированной программой (ФП). ФП собирается из готовых фрагментов вычислений. Выполнение ФП организуется с помощью специальной исполнительной run-time системы (ИС). Данная система исполняет фрагменты вычислений в порядке, не противоречащем отношению частичного порядка, и старается обеспечить динамические свойства ПП.

Цель работы

Целью работы было разработать модель визуализации исполнения фрагментированной программы и систему визуализации исполнения ФП, осуществляющую описанный в подразделе *«Результаты анализа»* централизованный real-time подход с сохранением порядка. Система должна наглядно отображать ход исполнения фрагментированной программы на мультикомпьютере.

Научная новизна данной работы состоит в разработке

1. визуального представления мультикомпьютера и исполняющейся фрагментированной программы;
2. алгоритмов учета событий в визуальной модели.

Требование поддержки real-time подхода важно, в свете того, что данная система в перспективе станет визуальным отладчиком, для которого такой подход единственный возможный.

Система визуализации должна отображать такую информацию как:

1. загруженность вычислительных узлов и коммуникационной сети кластера;
2. дисбаланс вычислительной загрузки между узлами;
3. промежутки времени простоя, ожидания и работы вычислительного узла;
4. сетевые взаимодействия;
5. издержки синхронизации частей программы.

Актуальность работы заключается в востребованности данной системы визуализации при разработке фрагментированных программ.

1. Визуальная модель исполнения фрагментированных программ

1. 1. Фрагментированное программирование

Технология фрагментированного программирования – это технология разработки параллельных программ, частично решающая проблемы разработки параллельных программ – программист освобождается от низкоуровневого программирования коммуникаций, синхронизации и реализации алгоритма балансировки нагрузки. ТФП направлена на то, чтобы автоматически обеспечить динамические свойства прикладной параллельной программы. Таким образом, фрагментированное программирование позволяет разрабатывать ПП с меньшими временными затратами и требует от программиста меньшей квалификации в области системного программирования.

Рассмотрим модель фрагментированной программы. Она состоит из двух частей.

Первая – «алгоритмическая» часть содержит представление алгоритма, ее можно описать кортежем $\langle D,F,<,in,out\rangle$, где

- D – множество фрагментов данных;
- F – множество фрагментов вычислений;
- $< \subseteq F \times F$ – отношение частичного порядка на множестве фрагментов вычислений;
- функции $in,out: F \to P(D)$, где $P(D)$ – булеан множества D, определяющие для каждого фрагмента вычислений множество соответственно входных и выходных фрагментов данных.

Фрагмент данных – это ячейка памяти, хранящая значения агрегированных переменных алгоритма, например, область памяти компьютера, в которой хранится часть сетки в конечно-разностном методе.

Фрагмент вычислений (ФВ) $f \in F$ задает некоторую составную операцию, вычисляющую значения выходных фрагментов данных $out(f)$ из входных $in(f)$.

Отношение частичного порядка $<$, обусловлено информационными зависимостями между фрагментами вычислений и требованиями к использованию ресурсов вычислительной системы. Данное отношение ограничивает порядок исполнения фрагментов вычислений, то есть, если верно $<(f_1, f_2)$, то фрагмент вычислений f_1 должен исполниться раньше f_2.

Специальная исполнительная run-time система (ИС) осуществляющая исполнение программы, запуская ФВ с порядке не противоречащем отношению $<$. Результатом работы ФП является состояние памяти компьютера на момент завершения программы. Для предотвращения ошибок одновременного чтения и записи в фрагмент данных ИС обеспечивает монопольный доступ на чтение и запись фрагментов данных $out(f)$ для исполняющегося в данный момент фрагмента вычислений f.

Фрагментированную программу можно представить в виде двудольного графа, в котором вершины представлены фрагментами вычислений и данных, а ориентированные ребра связывают ФВ с соответствующими им входными и выходными фрагментами данных. Например, алгоритм вычисления выражения $\sqrt{x^2 + y^3}$ может быть представлен фрагментированной программой $FP = \langle D,F,<,\text{in},out\rangle$, где $D = \{x, y, m_1, m_2, m_3, f\}$,
$F = \{f_1, f_2, f_3, f_4\}$, $< = \{(f_1, f_2), (f_2, f_3), (f_3, f_4)\}$;
$in(f_1) = \{x\}$, $in(f_2) = \{y\}$, $in(f_3) = \{m_1, m_2\}$, $in(f_4) = \{m_3\}$,

$out(f_1)=\{m_1\}, out(f_2)=\{m_2\}, out(f_3)=\{m_3\}, out(f_4)=\{r\}$; $f_1(x,m_1)=\{m_1 \neg x^2\}$, $f_2(y,m_2)=\{m_2 \neg y^3\}$, $f_3(m_1,m_2,m_3)=\{m_3 \neg m_1 + m_2\}$, $f_4(m_3,r)=\{r \neg \sqrt{m_3}\}$. Графическое представление FP изображено на рисунке 1.

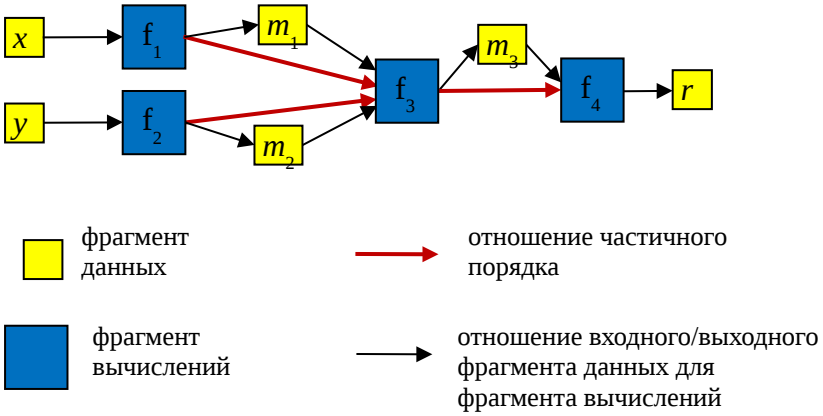

Рисунок 1 – Пример фрагментированной программы

Фрагменты могут быть агрегированными, то есть фрагмент данных может быть представлен массивом байт памяти, а фрагмент вычислений – составной функцией над входными ФД, например, запись в выходной ФД суммы двух векторов из входных или вычисление корней алгебраического уравнения.

Вторая часть модели фрагментированной программы содержит рекомендательную информацию, относящуюся к реализации алгоритма на конкретной вычислительной системе – информация об архитектуре мультикомпьютера, структурах данных и вычислений, вычислительной модели, рекомендуемом способе исполнения ПП и распределения данных и вычислений по узлам. Рекомендательная информация содержит основные решения, которые требуются для эффективного исполнения программы. Можно выделить рекомендации вида:

- $Nb \subset D \times D$ - отношение соседства на множестве фрагментов данных. ИС старается разметить соседние фрагменты данных на смежные вычислительные узлы;
- $Fc : F \to$ i - функция, оценивающую вычислительную сложность каждого ФВ. ИС перераспределяет фрагменты вычислений по узлам таким образом, чтобы вычислительная загрузка на каждом из них была примерно равной;
- $Fp : F \to$ ¥ - функция приоритетов ФВ, задающая рекомендуемый порядок исполнения, то есть при прочих равных условиях фрагмент вычислений с большим приоритетом выбирается первым для выполнения ИС.

Исполнительная система действует по следующему правилу – если для некоторого ФВ f все фрагменты вычислений из множества $\{p \mid <(p, f)\}$ уже выполнены, то f может быть назначен на исполнение. Множество готовых к исполнению ФВ может быть большим, поэтому возможен неудачный выбор очередного фрагмента вычислений на выполнение, например, последовательность неудачно выбранных ФВ может привести к нехватке памяти или снижению степени параллелизма. Для решения данной проблемы ИС руководствуется рекомендательной информацией, которая накладывает дополнительные ограничения на порядок исполнения ФВ. Таким образом, ИС осуществляет отображение ФП на ресурсы мультикомпьютера и исполняет ее в соответствии с имеющимся аппаратным обеспечением и заданными рекомендациями.

В ходе исполнения фрагментированная программа сохраняет фрагментированную структуру – множество фрагментов вычислений и данных, распределенных на узлах

мультикомпьютера. Таким образом, ИС может перемещать фрагменты с узла на узел, для обеспечения равномерной вычислительной загрузки. Например, в случае, когда много готовых к выполнению ФВ на текущем узле, который занят вычислениями, ИС пересылает ФВ на другой менее загруженный узел. Выбор фрагментов для миграции также определяется заданным программистом в рекомендательной информации ФП алгоритмом балансировки, например, миграция осуществляется из соображений минимизации количества межузловых ФВ $\leftrightarrow$ ФД связей.

В задачи программиста входит представление алгоритма решения задачи в фрагментированном виде – написание кода фрагментов вычислений, определение отношения частичного порядка, фрагментов данных и, если необходимо, задание рекомендательной информации. Отладка и верификация ФП включают в себя отладку отдельных фрагментов данных и доказательство корректности отношения частичного порядка.

1.2. Базовая визуальная модель

Система визуализации должна представлять состояние ИС в соответствие с некоторой визуальной моделью. Необходимыми требованиями для такой модели будет представление топологии и узлов вычислительной сети, распределенных по узлам объектов и связей между ними. При этом каждый узел, объект и связь обладают рядом свойств, изменяющихся во времени. В соответствии с требованиями была разработана описанная ниже визуальная модель.

Введем неориентированный граф $CL = (P, C)$ с множеством вершин $P = \{P_i\}_{i=1}^{n}$ и множеством ребер $C \subseteq (P \times P) = \{C_j\}_{j=1}^{m}$. Будем интерпретировать вершины как вычислительные узлы параллельного компьютера, а ребра – коммуникационные связи между ними. Каждую вершину и ребро графа CL можно представить в виде кортежа $P_i = \langle fp_1^i, fp_2^i, \ldots, fp_{fn}^i \rangle$, $C_j = \langle fc_1^j, fc_2^j, \ldots, fc_{fm}^j \rangle$, где $fp_k^i : T \to i$, $fc_k^j : T \to i$, $T = i^+ \cup \{0\}$, есть в общем случае вещественнозначные функции от времени, назовем их свойствами узлов и ребер (рисунок 2). Таким образом, множество значений этих свойств в определенный момент времени определяет состояние вершин и ребер, а значит, графа вычислителей в целом.

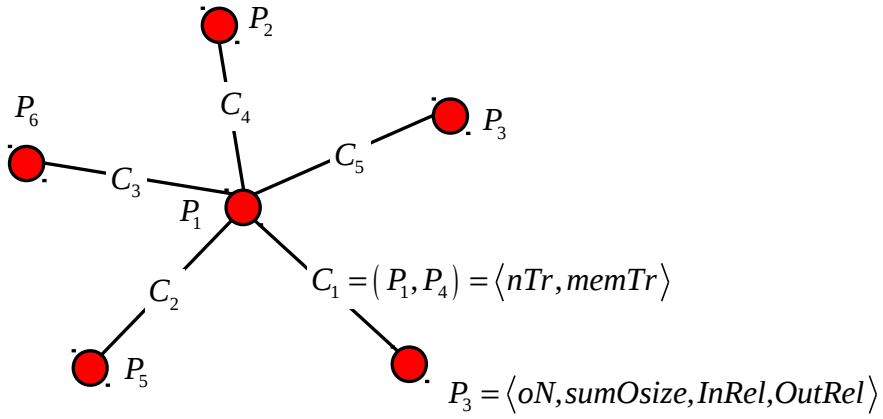

Рисунок 2 – Граф вычислителей

Введем неориентированный граф $Fg = (O, R)$ – множество объектов O, связанных некоторым отношением R. Назовем R отношением соседства, а объекты a и b, такие что $(a, b) \in R$, соседями. Определим функцию распределения объектов по узлам $\mathrm{Pos}: (\mathrm{O}, \mathrm{T}) \to P$, которая каждой вершине из O в определенный момент времени ставит в соответствие вершину графа CL.

Определим факт перемещения объекта с узла на узел как кортеж $\langle Send, \mathrm{Re}cv, Obj, Time \rangle$, где $Send, \mathrm{Re}cv \in P$ – посылающий и принимающий узлы соответственно, $Obj \in O$ – пересылаемый объект, $Time \in T$ – время совершения события. Таким образом, $\mathrm{Pos}(\mathrm{Obj}, \mathrm{Time}) = Send, \mathrm{Pos}(\mathrm{Obj}, \mathrm{Time}+) = \mathrm{Re}cv, (Send, \mathrm{Re}cv) \in C$. Пусть теперь $Tr = \{\langle Send_1, \mathrm{Re}cv_1, Obj_1, Time_1 \rangle, \ldots\}$ – множество кортежей, описывающих все событий пересылки объектов с узла на узел.

Каждая вершина Fg, объект, обладает свойством:

1. $\mathrm{Osize}: \mathrm{T} \to ¥$ – свойство, определяющее количество байт, которое занимает

объект в памяти вычислительного узла.

Для вершин P_k, $k=\overline{1..n}$ графа CL определим свойства:

1. количество объектов, расположенных на P_k: $\mathrm{nO(t)} = \mathrm{card}\left(\left\{ O_i \middle| Pos(O_i,t) = P_k \right\}\right)$;
2. объем памяти, занимаемый объектами на P_k:

$$\mathrm{sumOSize(t)} = \sum_{\mathrm{obj}\in\{o|o\in O\wedge Pos(o,t)=P_k\}} \mathrm{obj.Osize(t)};$$

3. число связей между объектами, расположенными на узле P_k:

$$\mathrm{InRel(t)} = \mathrm{card}\left(\left\{ (O_i,O_j) \middle| (O_i,O_j)\in R \wedge Pos(O_i,t)=P_k \wedge Pos(O_j,t)=P_k \right\}\right);$$

4. количество связей между объектами в P_k и другими вычислительными узлами:

$$\mathrm{OutRel(t)} = \mathrm{card}\left(\left\{ (O_i,O_j) \middle| (O_i,O_j)\in R \wedge Pos(O_i,t)=P_k \wedge Pos(O_j,t)\neq P_k \right\}\right).$$

Для ребер $\mathrm{C_j} = (S,R)$:

1. количество переданных по данной коммуникационной связи объектов:

$$nTr(t) = card\left(\left\{ tr \middle| tr\in Tr \wedge tr = \langle S,\mathrm{R},Obj,Time\rangle \wedge Time\le t \right\}\right);$$

2. количество данных в байтах переданных по связи $\mathrm{C_j}$:

$$memTr(t) = \sum_{trav\in\{tr|tr\in Tr\wedge tr=\langle S,\mathrm{R},Obj,Time\rangle\wedge Time\le t\}} Obj.Osize(Time).$$

Граф вычислителей и объектов можно отобразить визуально, как это представлено на рисунке 3.

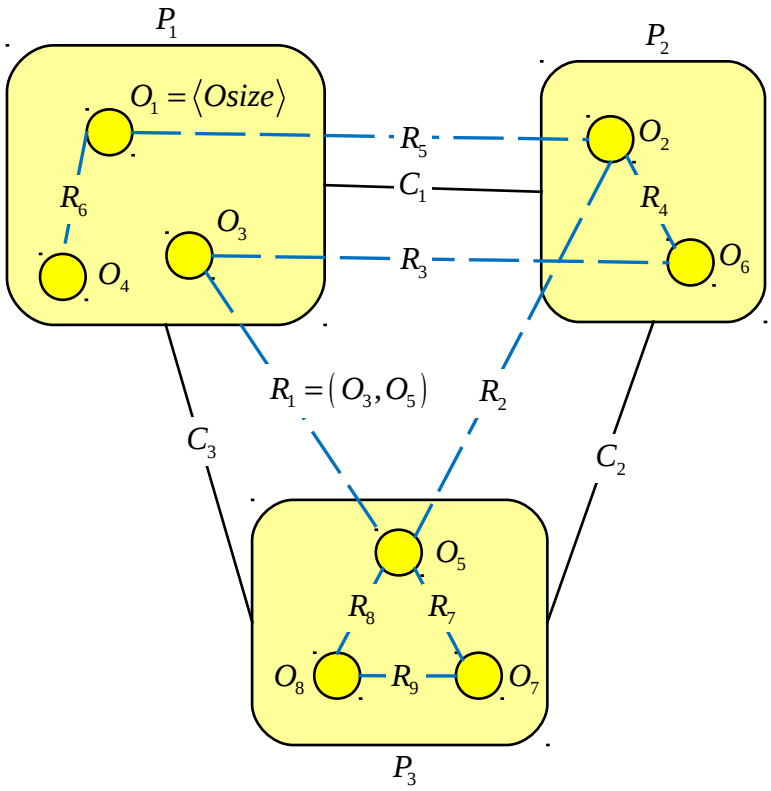

Рисунок 3 – Граф объектов, совмещенный с графом узлов

Описанная визуальная модель может быть применена для отображения исполнения фрагментированной программы при соответствующей адаптации модели, а именно, множество объектов разделится на два – $O = F \cup D$, где F, D – множество фрагментов вычислений и данных соответственно. Граф Fg станет двудольным ориентированным – $R \subseteq (F \times D) \cup (D \times F)$, таким образом, если $(f_i, d_j) \in R$, $f \in F, d \in D$, то $d_j \in out(f_i)$, соответственно, если $(d_j, f_i) \in R$, то $d_j \in in(f_i)$. Появится граф $Ctrl = (F, <)$, представляющий отношение частичного порядка $<$ на множестве ФВ. Визуальное представление графов CL, Fg и $Ctrl$ показано на рисунке 4.

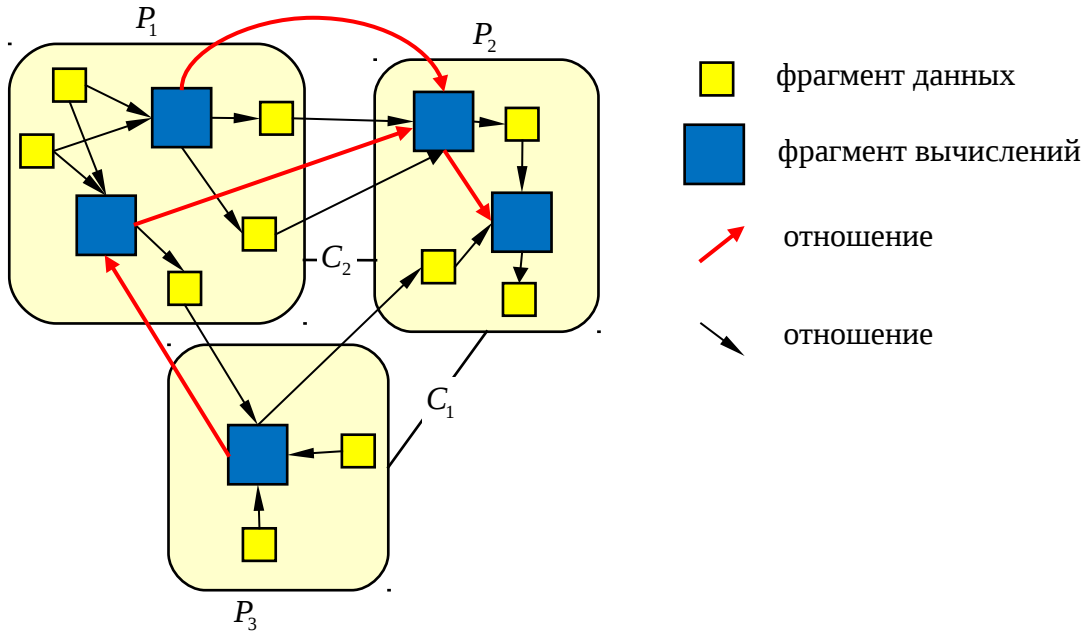

Рисунок 4 – Граф ФВ и ФД

1. 3. Модель вычислений в RFES4

RFES4 – это run-time система исполнения фрагментированных программ, разработанная в ИВМиМГ [12]. Рассмотрим применение этой визуальной модели для ИС RFES4. Опишем модель вычислений RFES4, чтобы потом соответствующим образом адаптировать базовую визуальную модель.

В исполнительной системе RFES4 понятие фрагмента вычислений и фрагмента данных объединено в одно – агент. Агенты распределены по вычислительным узлам и имеют жестко заданные коммуникационные связи между собой, в пределах которых они могут пересылать сообщения. Таким образом, в терминах визуальной модели множество агентов с заданным на нём отношением коммуникационной связи можно представить в виде графа $Fr = (O, R)$, а сеть вычислительных узлов в виде графа $CL = (P, C)$.

Агент начинает свое исполнение, когда получает от другого агента сообщение с определенными данными. В процессе исполнения агент может произвести вычисления, характер которых, вообще говоря, зависит от содержания полученного сообщения и отослать ряд сообщений другим агентам, после чего он завершает свое выполнение и переходит в режим ожидания следующего сообщения.

Исполнительная система создает агентов, организует их запуск и передачу сообщений между ними. Каждый агент, выполнив все необходимые вычисления, посылает ИС сообщение об окончательном завершении своей работы, после чего ИС удаляет агента,

освобождая занятую им память. Фрагментированная программа завершается, когда все агенты удалены ИС.

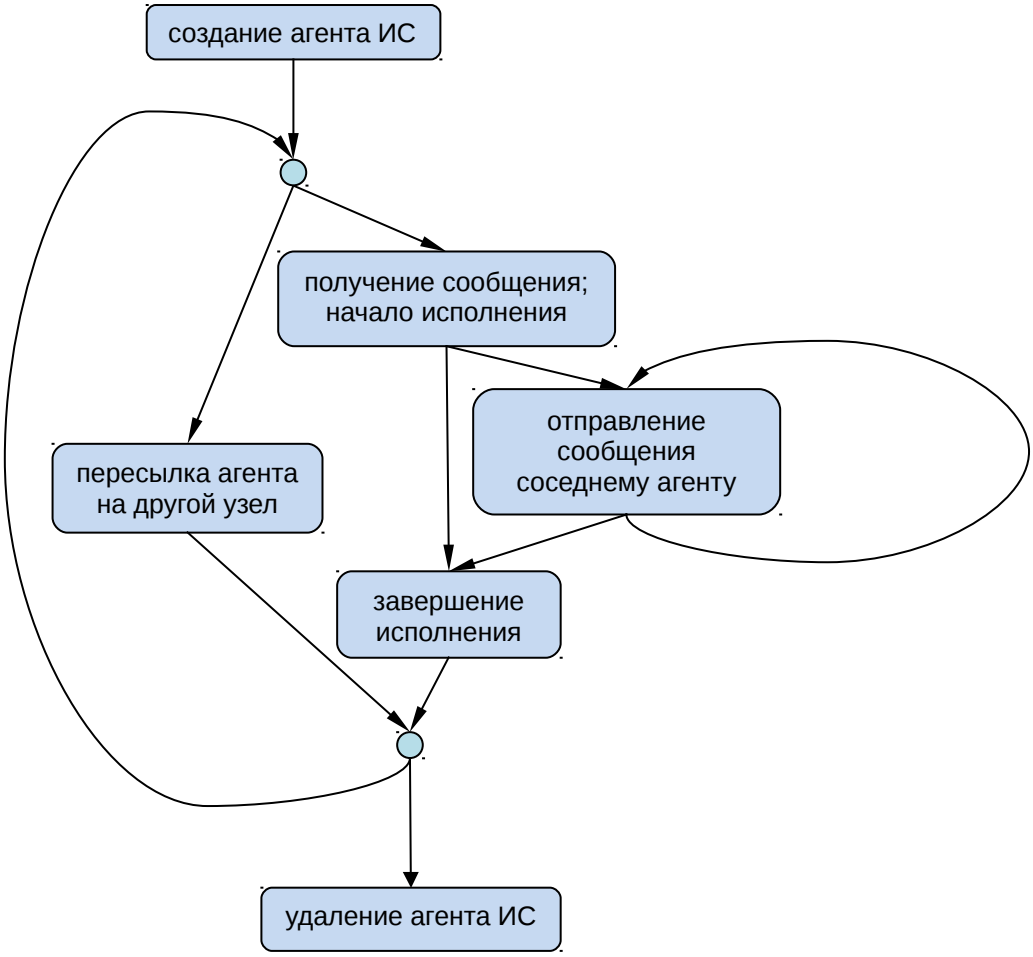

Рис. 5 – Граф событий жизненного цикла агента

1. 4. Адаптация визуальной модели к модели вычислений RFES4

Основной целью работы с системой визуализации является оценка эффективности исполнения ФП и выявление узких мест в работе программы и ИС. Требуется, чтобы СВ предоставляла информацию:

1. загруженность вычислительных узлов. Оценить загруженность можно с помощью таких свойств узлов, как:
   - количество одновременно исполняющихся на узле агентов;
   - число агентов на узле;
   - количество агентов в очереди на исполнение;
   - количество памяти, занятой всеми агентами, на узле;
2. загруженность коммуникационной сети кластера, которую можно аппроксимировать свойствами ориентированной коммуникационной связи между узлами:
   - количество сообщений в очереди на отправку на второй узел;
   - количество памяти, передаваемое по связи через сообщения между агентами;

Эффективность распределения исполнительной системой ресурсов по вычислительным узлам оценивается по максимальной вариации значений свойств, с помощью которых определяется загруженность элементов сети.

Опишем формально перечисленные свойства.

Будем считать, что момент времени $t = 0$ – точка отсчета времени работы исполнительной системы, то есть момент, когда ИС начинает свою работу хотя бы на одном из вычислительных узлов. Множество O – множество агентов. Введем объект, описывающий факт пересылки сообщения от агента к агенту – $\langle Send, Recv, Size, Time \rangle$, где $Send, Recv \in O$ – агент отправитель и приемник соответственно, $Size$ – размер посылаемого сообщения, $Time$ – время отправления. Множество всех событий посылки сообщений $msgTr = \{ \langle Send_1, Recv_1, Size_1, Time_1 \rangle, .. \}$.

Введем свойства для агента O_k:

1. $Cx : T \to \mathbb{i}$ – вычислительная сложность. В общем случае, это константа, определенная программистом при создании ФП, которая является оценкой времени исполнения агента;
2. количество запусков агента на исполнение: $nExec : T \to \forall$;
3. интервалы времени, в течение которых исполнялся агент: $execInt = \{ [startT_1, endT_1], \ldots, [startT_{nExec(t)}, endT_{nExec(t)}] \}$, где $startT_i, endT_i \in T, startT_i < endT_i, i = \overline{1..nExec(t)}$;
4. общая продолжительность исполнения агента – $ExecTime(t) = \sum_{[a,b] \in execInt, a<t} (\min(b,t) - a)$;

и для коммуникационной связи между агентами $R_k = (O_i, O_j)$:

1. $CommW \in \mathbb{i}$ – задаваемый программистом вес коммуникационного взаимодействия с другими агентами при посылке сообщения.

Определим ряд дополнительных свойств для вычислительного узла P_k:

1. количество одновременно исполняющихся на узле агентов: $nExecA(t) = card(\{ f_i \mid f_i \in F \wedge Pos(f_i, t) = P_k \wedge t \in f_i.execInt \})$;
2. длина очереди готовых к исполнению агентов на узле: $nQE : T \to \forall$;
3. сумма вычислительных весов всех агентов на узле: $sumCx(t) = \sum_{obj \in \{ o \mid o \in O, Pos(o,t) = P_k \}} obj.Cx(t)$.

Для коммуникационной связи $C_j = (S, R)$ введем свойства:

1. количество сообщений в очереди на отсылку с S на R: $nQS : T \to \forall$;
2. количество данных в байтах, переданных сообщениями с узла S на R: $msgMemTr(t) = \sum_{trav \in Tc} Size$, где $Tc = \{ tr \mid tr \in msgTr \wedge tr = \langle Send, Recv, Size, Time \rangle \wedge Pos(Send, Time) = S \wedge Pos(Recv, Time) = R \wedge Time \le t \}$.

Свойства вычислительных узлов, которые являются функциями от времени, можно отображать как двумерные графики с осями времени и значением свойства. Свойства коммуникационных связей между узлами в виде таблицы $n \times n$, где каждая ячейка отображает значение свойства для соответствующей связи за определенный промежуток времени. Статический граф вычислителей можно представить как это показано на рисунке 2,

а граф агентов с отображение положения агента в сети (функция Pos), изменяющегося во времени, – как на рисунке 3.

1. 5. Протокол отсылаемых ИС системе визуализации сообщений

Визуальная модель описывает, каким образом можно представить во времени состояние системы RFES, исполняющей ФП, и определяет свойства агентов и узлов, анализ которых может быть полезен в процессе отладки, нахождении узких мест и оптимизации ФП. Пусть состояние ИС есть функция от времени CD – совокупность всех графов, множеств и функций, определенных в визуальной модели. Требуется разработать такую систему, которая поддерживала бы представление CD в соответствии с визуальной моделью, взаимодействуя некоторым образом с ИС. Можно выделить как минимум два способа такого взаимодействия:

1. при изменении состояния ИС посылает системе визуализации сообщение, содержащее моментальный отпечаток (*snapshot*) состояния ИС – $CD(t_i)$. СВ дополняет представление ИС данным срезом.
2. ИС отсылает значение изменения своего состояния $\Delta CD = \frac{CD(t_i) - CD(t_{i-1})}{t_i - t_{i-1}}$.

   Система визуализации вычисляет $CD(t_i) = CD(t_{i-1}) + (t_i - t_{i-1})\,\Delta CD$.

   $CD(0)$ определяется как графы вычислителей и агентов без вершин и ребер.

В обоих случаях считается, что $\forall t \in [t_{i-1}, t_i)\ CD(t) = CD(t_{i-1})$.

Было решено использовать второй способ, так как ИС часто и незначительно меняет свое состояние. Таким образом, каждый экземпляр исполнительной системы, организующий исполнение ФП на соответствующем ему вычислительном узле, шлет системе визуализации сообщения, сигнализирующие о событиях – изменениях состояния ИС, произошедших на данном узле:

1. события, дающие представление о положении агента в сети вычислительных узлов, то есть такие события как:
   a. создание агента на узле исполнительной системой, обозначим данное событие как $ctorA$;
   b. пересылка агента с узла на узел – $sendA$;
   c. удаление ИС агента, т.е. освобождение занятой им памяти – $dtorA$;
2. изменение общего количества внутренних и внешних коммуникационных связей между агентами относительно данного узла – $relCnt$.
3. события, обозначающие изменение активности агента и посылки сообщений другим агентам:
   a. получение агентом сообщения, которое инициирует начало исполнения агента – $startA$;
   b. отправление сообщения от агента к агенту – $sendMsg$;
   c. завершение исполнения агента – $endA$.
4. изменение определенных величин в ИС, таких как
   a. длина очереди готовых к исполнению агентов – сообщение $Qexec$;
   b. длина очереди сообщений на отправку в другие узлы – $Qsend$.

События типа 1-3 происходят с некоторым агентом $O_i \in O$ на вычислительном узле $P_k \in P$ в определенный момент времени $t \in T$, таким образом, сообщение содержит в себе как минимум информацию $\langle O_i, P_k, t \rangle$.

Рассмотрим подробнее формат данных сообщений и то, каким образом система визуализации учитывает их при обновлении свойств графа вычислителей и агентов.

1. Каждое из событий $ctorA$, $sendA$, $dtorA$ можно разделить на последовательность элементарных событий – появление некоторого агента O_i на узле P_k и его исчезновение с узла. Описать такие элементарные события можно через кортеж с минимальной описанной выше информацией и свойствами агента, значение которых определяется программистом при написании ФП – $\langle O_i, P_k, t, \mathrm{Osize}(\mathrm{t}), Cx(t)\rangle$. Система визуализации доопределяет свойства графа вычислителей и агентов при получении события о появлении агента на узле следующим образом:
   - функция местоположения агента в вычислительной сети: $Pos(O_i, t) = P_k$;
   - количество агентов на узле: $\mathrm{nO(t)} = \mathrm{nO(t\text{-})}+1$;
   - сумма вычислительных весов агентов на узле: $sumCx(t) = sumCx(t-) + Cx(t)$;
   - объем памяти, занимаемый агентами: $sumOsize(t) = sumOsize(t-) + Osize(t)$.

   Подобным образом доопределяются свойства при исчезновении агента.
   Таким образом, при получении сообщения о событии перемещения агента с узла S на R, система визуализации обрабатывает его как события исчезновения с S и появления на R агента, кроме того обновляются свойства коммуникационной связи $C_i = (S, R)$:
   - число агентов, переданных по связи: $nTr(t) = nTr(t-) + 1$;
   - суммарное количество данных, переданных от S к R: $memTr(t) = memTr(t-) + O_i.Osize(t)$.

   Сообщение о создании агента имеет вид: $ctorA = \langle O_i, P_k, t, \mathrm{Osize}(\mathrm{t}), Cx(t), Nb\rangle$, где $Nb = \{o \mid o \in O \wedge R(o, O_i)\}$ – множество агентов, коммуникационно связанных с O_i, в соответствие с которым система визуализации устанавливает связи между агентами, т.е. формирует отношение соседства между агентами R графа Fg.
   В связи с возможностью нарушения хронологического порядка прихода сообщений, сообщение о перемещении или удалении агента может прийти в систему визуализации раньше сообщения о его создании, поэтому соответствующие сообщения должны содержать полную информацию о свойствах агента, значения которых используются при обновлении свойств графа вычислителей: $sendA = \langle O_i, P_k, R_k, t, \mathrm{Osize}(\mathrm{t}), Cx(t)\rangle$, где R_k – узел-приёмник агента; $dtorA = \langle O_i, P_k, t, \mathrm{Osize}(\mathrm{t}), Cx(t)\rangle$.
2. Учет системой визуализации свойств числа внутренних и внешних связей между агентами на узле – $InRel(t)$ и $OutRel(t)$, основываясь на сообщениях $ctorA$, $sendA$ и $dtorA$, затруднен в связи с тем, что из-за нарушения порядка получения сообщений сложно определить точное положения агента и его соседей в определенный момент времени. Поэтому было принято решение ввести сообщение RelCnt, которое генерируется исполнительной системой одновременно с сообщениями $ctorA$, $sendA$, $dtorA$ и несет в себе информацию о том, насколько изменились свойства $InRel(t)$ и $OutRel(t)$ для определенных узлов. Таким образом, при обработке сообщения $relCnt = \{t, \langle P_{\mathrm{i}_1}, inR_1, outR_1\rangle, \mathrm{K}, \langle P_{\mathrm{i}_{relSize}}, inR_{relSize}, outR_{relSize}\rangle\}$ обновляются свойства узла $\mathrm{P}_{\mathrm{i}_j}$, $j = \overline{1..relSize}$:
   - $\mathrm{InRel(t)} = \mathrm{InRel(t\text{-})} + inR_j$;
   - $\mathrm{OutRel(t)} = \mathrm{OutRel(t\text{-})} + outR_j$.

3. Формат данных для сообщений о начале и завершении исполнения агента одинаков: $startA = endA = \langle O_i, P_k, t, eventID \rangle$, где $eventID$ – уникальный идентификатор парных событий начала/завершения исполнения агента O_i.

   Система визуализации обновляет свойства узла P_k при получении сообщения $startA$:

   - количество одновременно исполняющихся на узле агентов: $nExecA(t) = nExecA(t-) + 1$;

   для агента O_i обновляются свойства:

   - множество интервалов времени исполнения агента $execInt$ – при получении пары сообщений $startA$ и $endA$ с одинаковым идентификатором $eventID$ система добавляет соответствующий интервал в множество $execInt$;
   - количество запусков – «срабатываний» агента O_i: $nExec(t) = nExec(t-) + 1$;

   Аналогично для сообщения $endA$.

   Прием сообщение $sendMsg = \langle O_i, P_k, t, R_j, msgSize \rangle$, где R_j – узел-приемник сообщения, $msgSize$ – размер сообщения в байтах, инициирует обновление системой свойства связи между узлами $C_l = (P_k, R_j)$, а именно, количество байт, переданных по C_l: $msgMemTr(t) = msgMemTr(t-) + msgSize$.
4. Последний тип сообщений, генерируемых ИС, содержит статистическую информацию о работе ИС – $Qexec = \langle P_k, t, nQexec \rangle$ и $Qsend = \langle P_k, R_j, t, nQsend \rangle$. Данные сообщения влияют на свойства:
   - длина очереди готовых к исполнению агентов на узле P_k, значение которой является оценкой вычислительной загруженности узла: $nQE(t) = nQexec$;
   - количество сообщений в очереди на отсылку– оценка коммуникационной загруженности связи $C_l = (P_k, R_j)$: $nQS(t) = nQsend$.

2. Реализация системы визуализации исполнения фрагментированных программ

Требуемая система визуализации для RFES4 должна наглядно отображать свойства узлов, агентов и связей между ними, предоставлять такие возможности как, просмотр списка событий ИС, получение детализованной информации, например, значение свойств ИС в определенный момент времени.

Предполагается использовать систему для ФП, исполняющихся на кластере под управлением *Linux*. Пользователь, работая на персональном компьютере вне сети кластера под управлением некоторой ОС (*Windows* или *Linux*), удаленно подключается к управляющему узлу кластера и запускает на выполнение ФП. Требует разработать систему визуализации, учитывая подобную специфику ее использования.

Как правило, кластеры состоят из одного управляющего узла и вычислительных узлов, объединенных между собой частной высокоскоростной сетью. Управляющий узел кластера выполняет функции сервера и шлюза с «внешним» миром. Этим обусловлено решение использовать клиент-серверную архитектуру.

Разработана система визуализации, состоящая из двух компонент. Первая – модуль для RFES4, реализующий сбор необходимой информации и отправку по разработанному протоколу сообщений, перечисленных в подразделе 1.5. «*Протокол отсылаемых ИС системе визуализации сообщений*» сообщений (*ctorA*, *sendA*, *dtorA*, *relCnt*, *startA*, *sendMsg*, *endA*, *Qexec* и *Qsend*). Вторая компонента – исполняющаяся на управляющем узле программа-визуализатор под названием FPVizor, которая принимает сообщения по разработанному протоколу от исполнительных систем с других узлов, вычисляет и отображает функцию *CD* (рисунок 6).

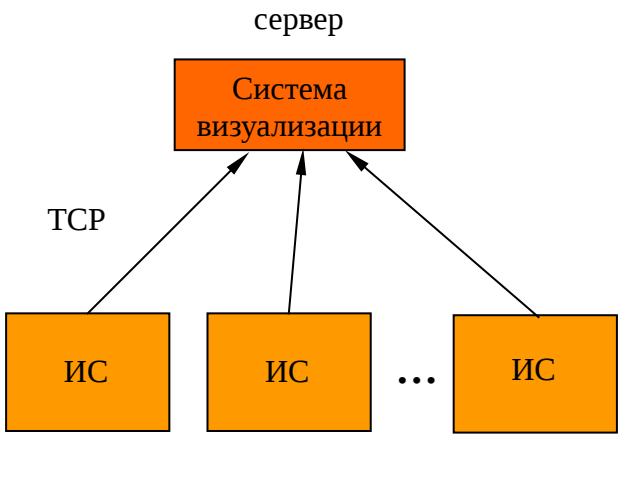

Рисунок 6 – Схема клиент-серверной архитектуры

Кластер работает под управлением серверной версии *Linux* без графического интерфейса. На управляющем узле есть библиотеки оконной системы *X Windows System*, что позволяет через компьютер под управлением *Windows* с помощью *Xming*-терминала подключиться к управляющему узлу и запускать на нем графические приложения, использующие *X*. При работе пользователя под управлением *Linux* достаточно выполнить команду

```
ssh -v <IP-адрес управляющего узла> -l <имя учетной записи пользователя> -X ,
```

далее, подключившись к узлу, запустить *xterm* – стандартный эмулятор терминала для среды *X Windows System* – и открыть программу-визуализатор FPVizor.

Для разработки графического пользовательского интерфейса FPVizor и работы с TCP-сокетами выбрана кроссплатформенная библиотека Qt [13]. Для построения графиков используется библиотека Qwt [14]. В качестве языка программирования выбран C++.

Модуль сбора и отправления сообщений для RFES4 представлен следующими классами:

TcpThreadClient

Класс, инкапсулирующий класс для работы с TCP-сокетами библиотеки Qt *QTcpSocket*. С помощью функции-члена класса *addMsg(const string&)* сообщение в виде массива байт записывается в очередь на отправление. Экземпляр класса работает в отдельном потоке – последовательно выбирает из очереди сообщения и отсылает их на указанный в конструкторе объекта класса порт и IP-адрес. Объект данного класса агрегируется классом *TcpLogger*.

TcpLogger

Экземпляр класса *TcpLogger* берет на себя функции отсылки сообщений. Конструктор *TcpLogger* принимает в качестве параметров IP-адреса компьютера и номер его порта, на который будут отсылаться сообщения. Для *TcpLogger* переопределен оператор `<<`, таким образом, чтобы можно было использовать запись `log << msg1 << msg2;`, означающую помещение сообщений *msg1*, *msg2* в очередь на отправку; *log* имеет тип *TcpLogger*, а *msg1* и *msg2* – экземпляры классов наследующих от *TcpReport*.

TcpReport

Абстрактный класс, представляющий общий интерфейс для всех передаваемых от ИС к СВ сообщений, а именно, функцию *print(std::ostream&)* для текстового представления содержания сообщения, функции сериализации и десериализации *serialize(GrowingObjectBuffer&)*, *deserialize(GrowingObjectBuffer&)*, где *GrowingObjectBuffer* – класс, инкапсулирующий массив байтов и функции добавления, извлечения из него объектов определенных типов данных (*int*, *double* и т.д.). Классы, наследуемые от *TcpReport*, такие, например, как, *TcpReportCtorF*, *TcpReportNeigRel*, соответствуют сообщениям $dtorA$ и $relCnt$.

Таким образом, сформированное ИС сообщение сериализуется, то есть представляется в виде последовательности байтов $SrMsg$, и отправляется на управляющий узел вместе с значением размера сообщения – $\langle sizeof(SrMsg), SrMsg \rangle$. Система визуализации принимает данные через TCP-сокет, считывает размер сообщения $sizeof(SrMsg)$ и, если $sizeof(SrMsg)$ байт доступно для чтения из сокета, считывает само сообщение и десериализует массив байтов в сообщение, которое потом обрабатывается FPVizor. Работа с сокетами в FPVizor возложена на объект класса *TcpServerViz*, который помимо приема данных из TCP-сокета предоставляет возможность чтения сообщений из файла.

Каждое приходящее программе-визуализатору сообщение содержит в себе значение момента времени t, в который произошло событие, описываемое сообщением. При получении сообщения система обновляет значение временного фронта tF – максимальное время t среди всех пришедших сообщений. Если приходит сообщение с временной отметкой $tLast$ меньшей, чем tF, значит нарушен хронологический порядок прихода сообщений и необходимо обновить значение всех свойств в моменты времени после tF. Например, если пришло сообщение о создании агента на узле P_k в момент времени $tCtor$, то FPVizor вставляет сообщение в соответствующее место в сохраненном списке событий и добавляет

единицу ко всем значениям свойства количества агентов для узла P_k в моменты времени после *tCtor* . Аналогичный учет происходит с остальными свойствами.

Так как сообщения могут приходить не в хронологическом порядке, возможно отклонение значений свойств от их действительных значений. В то же время, сохраняется определенный баланс для свойств узлов, например, количество агентов на узле в начале работы ИС и в конце равно нулю, так как общее число появлений агентов равно числу их удалений для конкретного узла на момент завершения работы ИС. Свойство нулевого баланса сохраняется, если такие свойства агента как вычислительный вес и занимаемый им объем памяти не зависят от времени, то есть являются константами.

2. 1. Ход работы с программой

Для работы в режиме приема сообщений от ИС с узлов вычислительной сети и отображения представления состояния ИС одновременно с исполнением ФП пользователю необходимо произвести следующие действия:

1. подключиться через Xming-терминал к управляющему узлу и запустить программу-визуализатор FPVizor (рисунок 7);

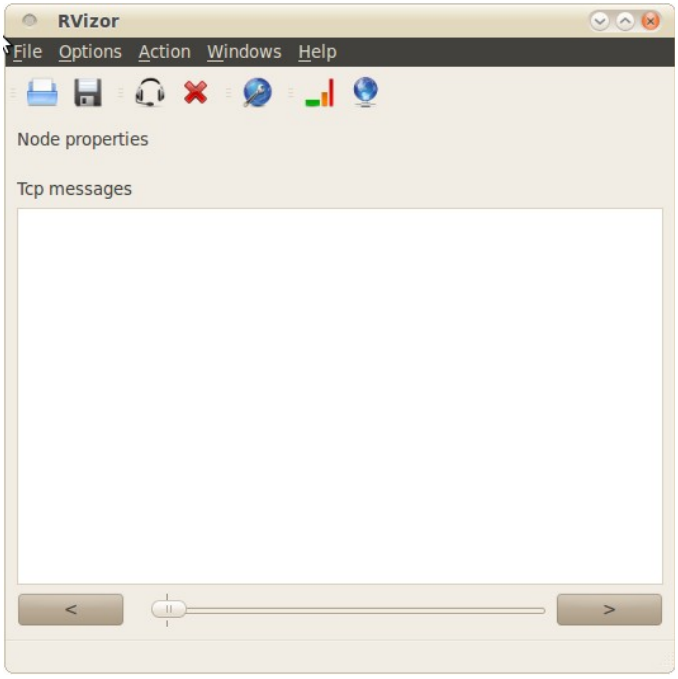

Рисунок 7 – Главная форма визуализатора FPVizor

2. указать в диалоговом окне настроек *Settings*, открываемом через меню *Options* → *Settings…* или нажатием соответствующей кнопки, номер прослушиваемого на управляющем узле порта (рисунок 8);

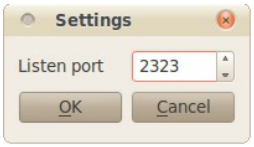

Рисунок 8 – Диалоговое окно *Settings*

3. нажатием на кнопку *Listen* или выбором команды через меню *Action* → *Listen* перевести программу в режим прослушивания установленного в пункте 2. порта;

4. запустить RFES4 через менеджер MPI-процессов *mpiexec* с указанием в параметрах командной строки IP-адреса управляющего узла и номера порта, на который ИС будет отправлять сообщения, например:
   ```
   mpiexec –n 4 ./rfes4 <ряд параметров, передаваемых ФП>
   198.162.0.199 2323
   ```
   Параметр *–n* задает количество MPI-процессов, менеджер *mpiexec* распределяет процессы по узлам в соответствие с имеющимися вычислительными ресурсами, в свою очередь, система визуализации считает каждый MPI-процесс отдельным вычислительным узлом.
5. пронаблюдать за исполнение ИС и, если необходимо, сохранить последовательность полученных сообщений в файл с расширением *fpv* с помощью команды *File → Save file…*. Такие файлы можно открыть через команду *File → Open file…*, тогда FPVizor прочтет все сообщения так, как если бы получала их от ИС по TCP.

Принимая сообщения от RFES4, программа-визуализатор обрабатывает их, обновляя представление описанной ранее визуальной модели. FPVizor добавляет в список событий *Event list* текстовое описание события, например, для события $ctorA = \langle (1,1), 2, 200, 24, 30.0, \{(1,0), (2,0)\} \rangle$ выводит строку

«в момент времени 200 ms. на узле ID: 2 создан агент ID: (1,1); размер 24 байт; выч. сложность: 30.0; соседи ID: [(1,0), (2,0)]»,

и обновляет значение полей *Node properties* на главной форме, отображающих свойства соответствующего вычислительного узла – $oN, sumOsize, InRel, OutRel, nExec, nQE, sumCx$ (рисунок 9). Изменение положения слайдера *Time Slider* на главной форме приложения устанавливает представление узлов в момент, соответствующий положению слайдера.

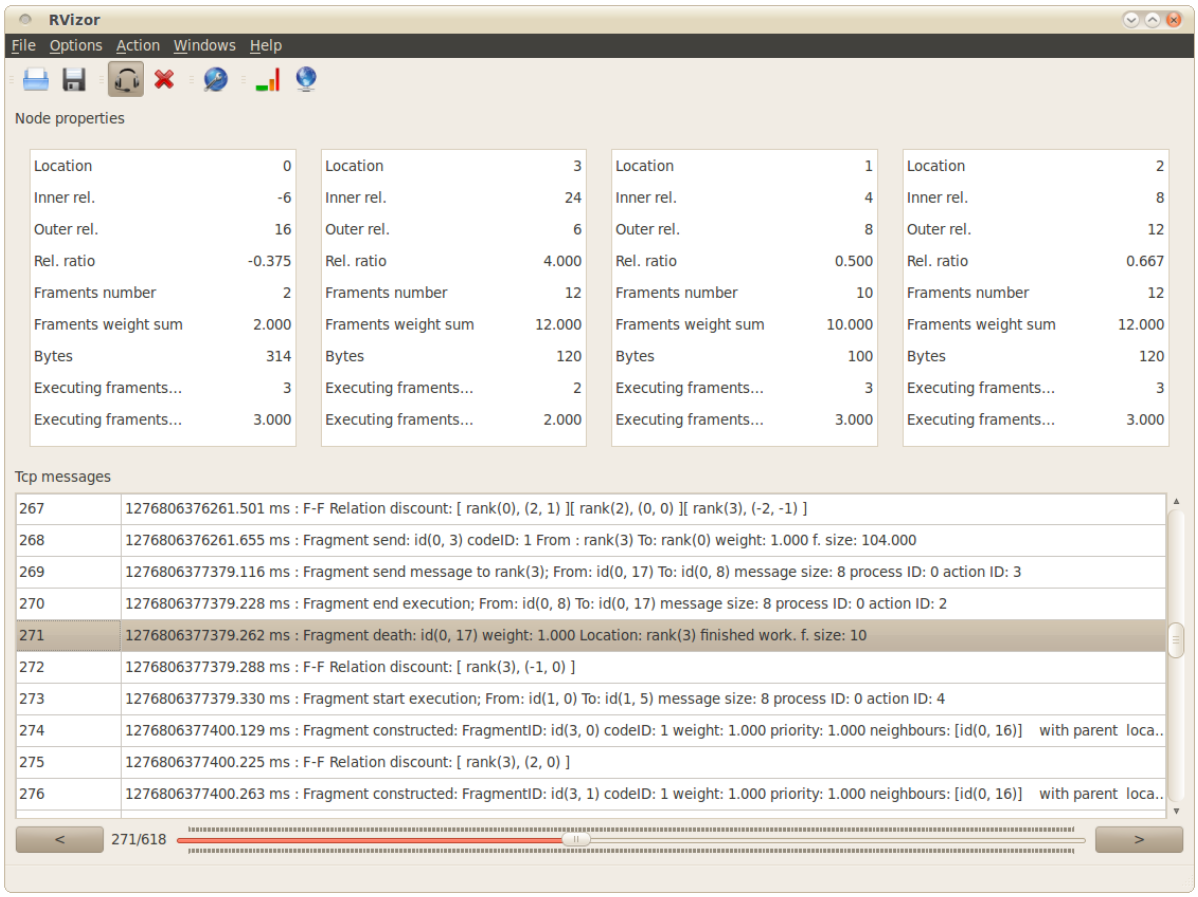

Рисунок 9 – FPVizor, получивший сообщения от RFES

В FPVizor присутствует возможность отображения перечисленных свойств узла в виде двумерных графиков от времени, которые отображаются в окне *Graphic*, открываемом нажатием на кнопку *Graphic* или через меню *Window* → *Graphic* (рисунок 10). Отображение определенного графика включается соответствующими чекбоксами. В окне присутствуют два бегунка для навигации по временной прямой. Нижний слайдер отвечает за начало временного интервала, слайдер справа – за длину интервала.

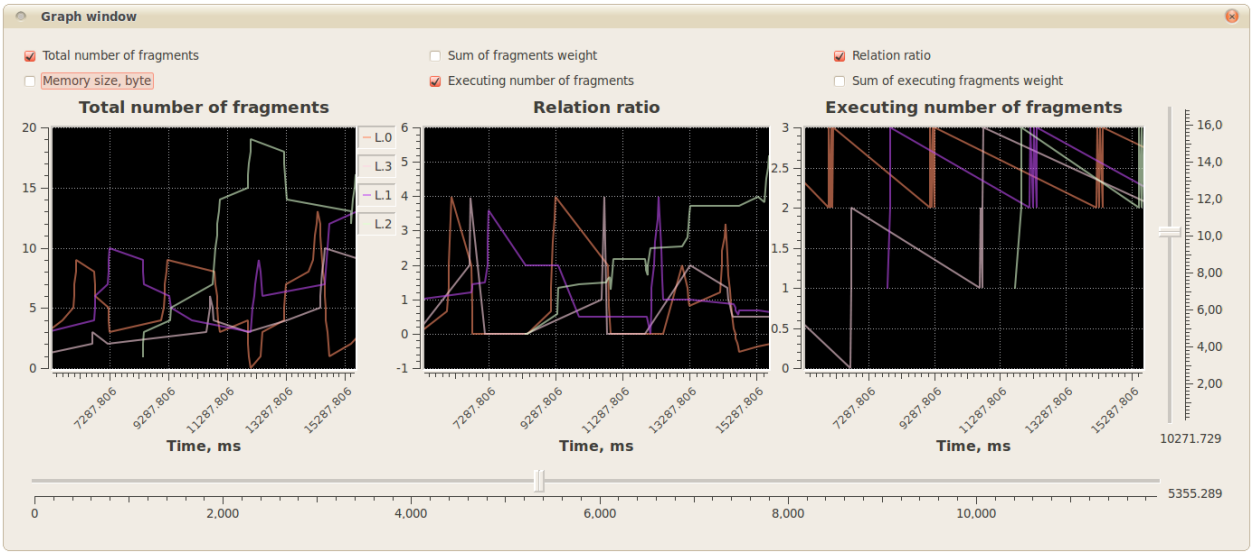

Рисунок 10 – окно *Graphic*

Свойства коммуникационных связей между узлами nQS, $msgMemTr$ отображаются в виде коммуникационной матрицы в окне *Communication matrix*, открываемом через команду *Window* → *Communication matrix* (рисунок 11). Коммуникационная матрица представлена в виде квадратной таблицы $n \times n$, где n – количество вычислительных узлов, (i, j) -ячейка таблицы отображает значение свойства связи $C_k = (P_i, P_j)$ и окрашена в цвет, зависящий от величины свойства: для наибольшего значения в таблице – красный цвет, для наименьшего – черный. Элементы управления те же, что и в окне *Graphic*.

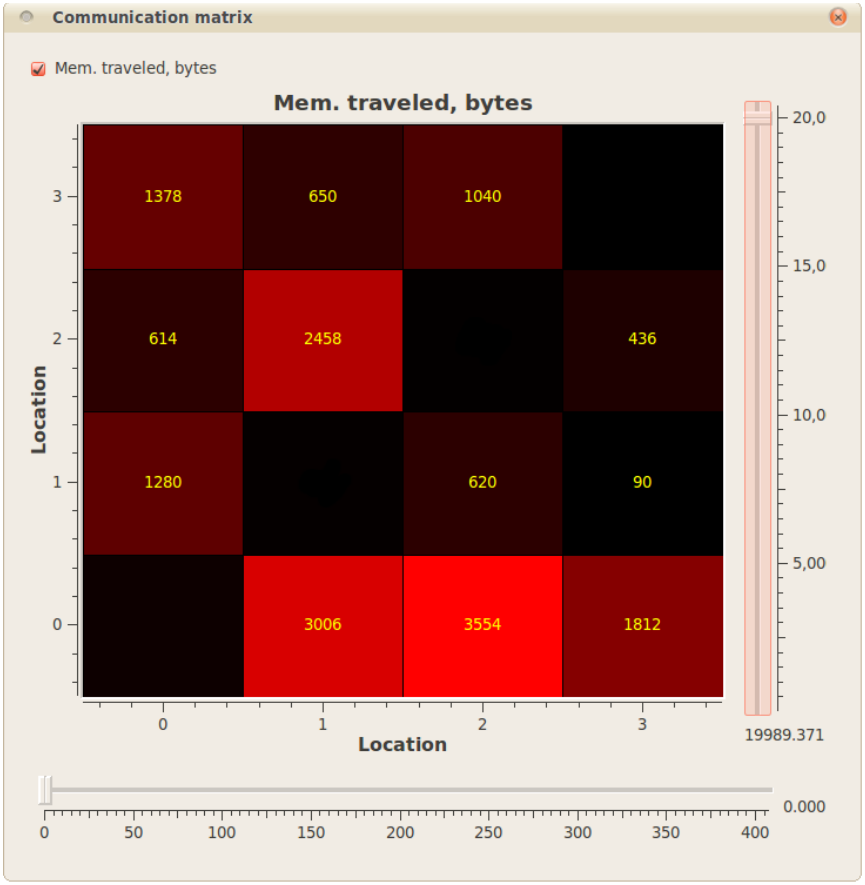

Рисунок 11 – окно *Communication matrix*

Таким образом, свойства отображаются СВ как в отдельный момент времени (поля *Properties*), так и в определенном временном интервале (окна *Graphic* и *Communication matrix*).

2. 2. Особенности и ограничения реализации

Данная реализация предъявляет требование наличия у пользователя свободно распространяемого Xming-терминала, библиотек *X Window System* и одной из реализаций MPI на управляющем узле.

Так как FPVizor и RFES4 являются кроссплатформенными программами и существуют реализации MPI под *Windows*, систему визуализации можно использовать в кластерах под управлением *Windows*. Использование разработанной СВ возможно на разных вычислительных системах, в силу того, что MPI-программы можно запускать как на системах с распределенной памятью (MPP-системы, кластеры), так и с общей (многопроцессорные компьютеры с SMP или NUMA архитектурой).

Не реализовано отображение графов вычислителей и агентов.

Система визуализации считает MPI-процесс отдельным вычислительным узлом, что, вообще говоря, не верно, например, при запуске нескольких MPI-процессов на одном узле, но эта особенность является вполне приемлемым ограничением.

Заключение

В дипломной работе приведен анализ существующих средств визуализации исполнения параллельных программ, изучен теоретический материал – методы визуализации исполнения параллельных программ и основы технологии фрагментированного программирования. Разработана визуальная модель представления исполняющейся фрагментированной программы. Система run-time визуализации фрагментированных программ, состоящая из модуля для исполнительной системы RFES4 и программы-визуализатора FPVizor. Проведено начальное тестирование, которое подтвердило возможность практического использования разработанной системы.

На защиту выносятся следующие пункты:

- визуальная модель представления состояния исполнительной системы;
- протокол передачи информации об изменении состояния исполнительной системы;
- алгоритмы и реализация системы визуализации исполнения фрагментированных программ.

Работу планируется продолжить в магистратуре в следующих направлениях:

- усовершенствование системы
  - добавление возможности отображения графа вычислительных узлов;
  - отображение актуального графа агентов, что требует активного разрешения проблемы нарушения порядка получения сообщений от ИС;
  - обеспечение поддержки работы с другими исполнительными системами;
  - сбор и отображение статистики, связанной с пользовательскими сообщениями, генерируемыми агентами в определенных обстоятельствах;
  - визуализация коэффициента использования кластера, интервалов времени исполнения агентов;
- разработка визуального отладчика фрагментированных программ, позволяющего
  - останавливать и запускать выделенные пользователем агенты;
  - устанавливать точки останова в коде агентов;
  - отображать и изменять значения доступных отладчику локальных переменных агента в точках останова.

Список использованных источников

1. Grama, A. Introduction to Parallel Computing / A. Grama [и др.]. – Second Edition. – Addison Wesley, 2003. – 856 с.
2. Малышкин, В.Э. Параллельное программирование мультикомпьютеров. / В.Э. Малышкин. – Новосибирск : Изд-во НГТУ, 2006. – 296 с.
3. TotalView debugger [Электронный ресурс]. – Режим доступа: http://www.totalviewtech.com/products/totalview.html.
4. Etnus Announces TotalView for IBM Blue Gene/L [Электронный ресурс]. – Режим доступа: http://www.hpcwire.com/offthewire/17870824.html.
5. Jumpshot-4 User's Guide [Электронный ресурс] : рук. пользователя. – Режим доступа: http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/jumpshot-4/usersguide.html.
6. ParaGraph: A Performance Visualization Tool for MPI [Электронный ресурс] : рук. пользователя. – Режим доступа: http://www.csar.illinois.edu/software/paragraph/userguide.pdf.
7. Rivet: A Flexible Environment for Computer Systems Visualization [Электронный ресурс]. – Режим доступа: http://www.siggraph.org/publications/newsletter/v34n1/contributions/Bosch.html.
8. Projections Manual [Электронный ресурс] : рук. пользователя. – Режим доступа: http://charm.cs.uiuc.edu/manuals/pdf/projections.pdf.
9. Intel Thread Profiler 3.1 for Windows – Guide to Sample Code [Электронный ресурс]. – Режим доступа: http://software.intel.com/en-us/articles/intel-thread-profiler-for-windows-documentation/.
10. Tomas, G. Visualization of Scientific Parallel Programs / G. Tomas, C. W. Ueberhuber. – Berlin Heidelberg : Springer-Verlag, 1994. – 310 с.
11. Вальковский, В.А. Синтез параллельных программ и систем на вычислительных моделях. / В.А. Вальковский, В.Э. Малышкин. – Новосибирск : Наука, 1988. – 128 с.
12. Malyshkin, V. E. Run-time system for parallel execution of fragmented subroutines. / V. E. Malyshkin [и др.] // Proceedings of the 9th International conference on Parallel Computing Technologies. – 2007. – Vol. 4671. – С. 544-552.
13. Qt Reference Documentation [Электронный ресурс]. – Режим доступа: http://doc.trolltech.com/4.6/index.html.
14. Qwt User's Guide [Электронный ресурс] : рук. пользователя. – Режим доступа: http://qwt.sourceforge.net/.

Приложение: Текст программы

Здесь будут фрагменты кода программы.