УДК 519.685

СИСТЕМА АВТОМАТИЗАЦИИ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ GRAPHPLUS TEMPLET

С.В. Востокин
Самарский государственный аэрокосмический университет им. академика С.П. Королева
Россия, 443086, Самара, Московское шоссе, 34
E-mail: easts@mail.ru

Ключевые слова: средство разработки, параллельное программирование, модель вычислений
Key words: software development tool, parallel programming, computational model

Описана новая система автоматизации параллельного программирования GraphPlus templet. Рассмотрены цели проектирования, архитектура системы, пример программирования, особенности системы в сравнении с аналогичными подходами.
PARALLEL PROGRAMMING AUTOMATION SYSTEM GRAPHPLUS TEMPLET / S.V. Vostokin (Samara State Aerospace University, 34 Moskovskoe shosse, Samara 443086, Russia, E-mail: easts@mail.ru). The article describes new parallel programming automation system named Graphplus templet. Design goals, the system architecture, sample program, the system features in comparison with analogs were discussed.

1. Введение

В работе представлена система автоматизации параллельного и распределенного программирования Graphplus templet, разработанная в рамках исследовательского проекта «Граф Плюс» (graphplus.ssau.ru) Самарского государственного аэрокосмического университета (национального исследовательского университета).

Последние 10 лет развития компьютерной индустрии характеризуются широким использованием параллельных вычислительных архитектур, как в традиционной области высокопроизводительных научных вычислений, так и в коммерческом, потребительском секторе (например, параллельные методы обработки стали применяться в web-обозревателях). Наряду с доступностью и широтой применения параллельных архитектур можно отметить также и их разнообразие: многоядерные настольные системы; кластерные системы и суперкомпьютеры; распределенные системы в сети интернет; процессоры CUDA; специализированные ПЛИС. Развитие аппаратных архитектур порождает спрос на программное обеспечение, позволяющее эффективно использовать новые параллельные вычислительные ресурсы.

В настоящее время отмечается, что спрос на параллельное программное обеспечение остается в одинаковой степени не удовлетворенным среди пользователей коммерческих приложений и среди исследователей, желающих применять методы параллельных вычислений в своих предметных областях. Так во многих популярных коммерческих продуктах пока не используются все возможности многоядерной обработки, а сфера применения параллельной обработки в научных вычислениях, по оценкам аналитиков, может быть увеличена в несколько раз по сравнению с текущим состоянием. Таким образом, актуальность задач, связанных разработкой параллельного программного обеспечения, за последнее время только увеличилась.

2. Метод автоматизации. Цели разработки

На основании известного и собственного опыта параллельного программирования можно выделить следующие актуальные проблемы, на преодоление которых направлена предлагаемая система автоматизации программирования.

Практика показывает, что программист средней квалификации обычно не может самостоятельно разработать параллельный алгоритм решения задачи. Более того, при наличии точного описания алгоритма возникают проблемы его реализации требуемыми средствами. Это кардинально отличается от последовательного программирования, где процесс перехода от спецификации к реализации в настоящее время достаточно формализован. Поэтому разработка параллельных программ должна быть основана на максимальном повторном использовании имеющихся решений. Известно около двух десятков типовых решений в области параллельного программирования. Требуется предложить способ удобного описания и конструирования программ на их основе, пригодный для автоматического использования. Причем такие типовые решения должны быть по возможности языково-нейтральными и предусматривать реализацию кода на языках, используемых в коммерческой разработке Java, C#, C++.

Существует ряд проблем, обусловленных неудобством базовых средств программирования параллельных вычислений. В первую очередь - надежность кода. Известно, что недетерминированный характер вычислительного процесса затрудняет выявление ошибок методами отладки. Также возникают трудности при переносе программ в различные параллельные архитектуры. Многопоточные программы обычно не удается сделать распределенными, если это не предусматривалось изначально при проектировании. Представляет большой интерес и разработка масштабируемых программ, не теряющих эффективность при увеличении аппаратных ресурсов. Общий подход в решении перечисленных проблем – использование формальной модели вычислений, которая позволяет выполнить строгую декомпозицию кода, более удобную по сравнению с аппаратно-ориентированной моделью многопоточности.

При наличии понимания особенностей параллельного алгоритма, методов его программирования, встает вопрос стоимости и сроков разработки программного продукта. Современные программисты привыкли работать в интегрированных средах (IDE). Процесс разработки и отладки параллельных программ в настоящее время отличается более низкой автоматизацией. Очевидный способ решения этой проблемы – встраивание системы параллельного программирования в известные процессы разработки программного обеспечения и интегрированные среды разработки, поддерживающие их. То есть такая система должна быть реализована как стандартное расширение известной среды разработки. В противном случае ее практическое использование становится проблематичным. Отладка также должна выполняться стандартными средствами среды разработки в последовательном режиме.

Таким образом, проектирование системы Graphplus templet выполнено с учетом следующих требований: (1) возможность работы с типовыми решениями на разных языках программирования (C++, C#, Java); (2) независимость прикладного кода, разрабатываемого в системе, от API управления вычислениями, а также взаимозаменяемость API без переработки приложений (например, API многопоточности Win32 на POSIX Threads); (3) совместимость с коммерческими интегрированными средами разработки (например, IDE Microsoft Visual Studio), в частности, с целью удобства отладки параллельных приложений стандартными средствами IDE.

К классам приложений, на которые в первую очередь ориентирована система, относятся приложения для научных вычислений (численного моделирования), оптимизированные для работы как в распределенной, так и разделяемой памяти. Система также предназначена для разработки прикладных протоколов асинхронного взаимодействия в коммерческих распределенных приложениях.

В работе описывается новая архитектура системы Graphplus templet и рассматривается, каким образом выбор архитектуры обеспечивает удовлетворение перечисленных требований к системе.

3. Модель программирования

Семантическая основа языка системы Graphplus templet - модель процессов диффузного типа. В данной модели конструктивными элементами являются пассивные объекты – *процессы* и активные объекты – двунаправленные *каналы*, соединяющие их. Объекты выполняют свои методы при поступлении сообщений по каналам и в ответ могут сформировать произвольное число сообщений (от 0 до некоторого N).

Как процессы, так и каналы детализируются и могут быть визуализированы при помощи аннотированных ориентированных графов специального вида. Цель такой детализации для процессов – выделить в них последовательные процедуры (*методы*), определить порядок их запуска при поступлении сообщений, а также описать информационный контекст методов процесса. Примеры применяемых для описания процессов пометок дуг и вершин графов показаны на рис. 1.

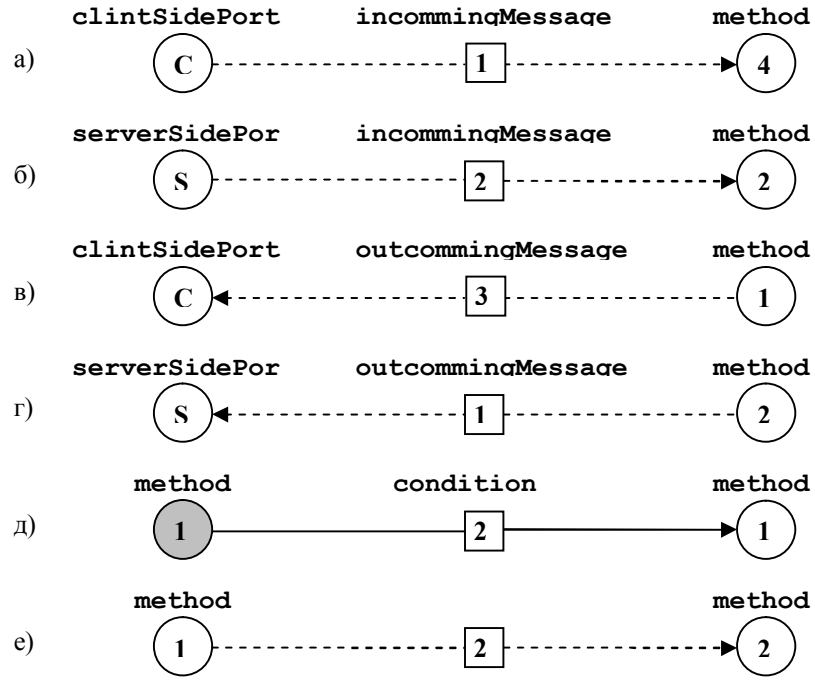

Рис. 1. Графические символы, используемые для описания процессов.

В графическом описании используются три типа вершин: (c) – клиентская сторона подключения канала, называемая клиентским портом; (s) – серверная сторона подключения канала или серверный порт; (1..n) – методы процесса. Вершины связываются дугами двух типов. Пунктиром обозначены дуги, описывающие передачу или прием сообщений. Пометки пунктирных дуг – это имена сообщений. Сообщения поступают или передаются в каналы через порты. Кроме этого (вариант е) сообщения могут передаваться процессом самому себе. Этот прием используется для удобства моделирования активного процесса в диффузной модели, где процессы считаются пассивными. Метод, отмеченный серой заливкой (вариант д) требуется для запуска вычислений в начальном состоянии. Сплошные дуги обозначают условную передачу управления. Пометки таких дуг – условия, вычисляемые на состоянии процессов. Цифры внутри методов используются для синхронизации. Поток управления

попадает в процесс с поступающим сообщением. Каждый раз, когда он достигает метода, происходит увеличение внутреннего счетчика, предельное значение которого обозначает цифровая метка. Метод запускается, когда счетчик достигнет предельного значения. При этом счетчик обнуляется, а поток управления движется далее по исходящим из метода дугам. Цифровые метки дуг обозначают их приоритет. Более подробно смысл обозначений рассмотрен в приведенном ниже примере.

Цель графического представления каналов - строго описать протокол взаимодействия процессов, соединенных ими. Также такая структура каналов неявно вводит ограничения, упрощающие реализацию модели в известных архитектурах. Граф канала – это конечный детерминированный автомат, в котором вершины являются состояниями передачи, а дуги являются сообщениями, разрешенными для передачи в инцидентном состоянии. На рис. 2 представлены графические обозначения, используемые для описания каналов.

Вершины в приведенном описании обозначают состояния канала. В состояниях с пометками (c) или (s) разрешается передача из, соответственно, клиентского или серверного порта канала. Сообщения, передача которых разрешена в данном состоянии канала, отмечают исходящие дуги. Когда процесс выбрал сообщение для передачи, канал переходит в новое состояние, причем серверные и клиентские состояния должны чередоваться. Исходным состоянием канала является некоторое клиентское состояние, обозначенное серой заливкой (случай а). Конечное состояние (если таковое имеется в протоколе канала) - это вершина, не имеющая исходящих дуг.

Клиентские и серверные порты подключения каналов в процессах однозначно связаны с типами каналов. Это исключает связь процессов через канал, протокол которого не поддерживают одновременно два связываемых процесса.

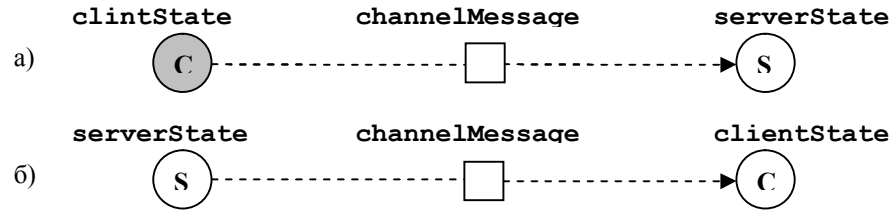

Рис. 2. Графические символы, используемые для описания каналов.

Еще одной дополнительной абстракцией модели программирования Graphplus templet является *сборка*. Сборка - это набор типов (разновидностей) процессов, каналов, а также других сборок. Со сборкой может быть ассоциирована конкретная топология процессов/каналов, а также специализированный для этой топологии механизм исполнения.

Графическое представление процессов и каналов в данной модели исполнения существенно упрощает их понимание. Необходимость такого представления обусловлена тем, что контекст вызова метода процесса трудно свести к одному предшествующему и одному последующему методу, как в последовательном структурном программировании. Это вызвано естественной для параллельного исполнения причиной: некоторому методу может предшествовать исполнение нескольких параллельных во времени действий, а несколько параллельных во времени действий могут инициироваться при завершении метода. Поэтому и существует проблема представления контекста вызова в виде линейного текста, решаемая путем использования графического представления.

Другие аспекты «многомерности» параллельных программ, например, связи между процессами, значительно проще представляются в последовательном коде. В настоящее время в системе Graphplus templet для сборок не используется графическое представление. Порядок соединения процессов между собой достаточно удобно описывается

последовательным алгоритмом, так как такие соединения обычно имеют регулярную или простую структуру.

4. Связь кода и модели программирования. Процесс разработки

Особенностью предлагаемого подхода является то, что семантика модели программирования передается не на специальном языке, а на традиционном языке (C++, C#, Java) без использования примитивов для задания параллелизма. Это делается следующим образом.

Предполагается, что весь код программы разбит на модули. Каждый модуль представляет собой один или несколько файлов. В программе различаются модули, код которых интерпретируется в терминах описанной вычислительной модели. Интерпретируемость означает, что в коде можно выделить участки, соответствующие элементам модели: процессам, каналам, клиентским и серверным портам, состояниям, методам, условиям, сборкам. Такие модули имеют строго определенную структуру, нарушать которую запрещено. Также в программе имеется специальный модуль с предопределенной структурой, описывающий механизм исполнения модели. Исполнение программы является недетерминированным. Наличие нескольких возможных вариантов развития вычислительного процесса передается в модуле механизма исполнения с использованием генератора случайных чисел. Остальные модули могут содержать код произвольной структуры. Таким образом, программа в терминах модели исполнения является последовательной, построенной специальным образом с учетом ограничений на структуру кода и передает функциональные аспекты поведения аналогичной параллельной программы.

Далее требуется определить два алгоритма: алгоритм проверки интерпретируемости программы в терминах модели; алгоритм распараллеливания – преобразования исходной последовательной программы в параллельную программу с теми же функциональными свойствами. Для этого вместо процедур лексического и синтаксического анализа используется процедура генерации. Каждый интерпретируемый модуль снабжается XML-спецификацией, описывающей объекты модели программирования, содержащиеся в нем. Код модуля генерируется по этой спецификации. При этом код состоит из чередующихся фрагментов – кусков кода, соответствующих элементам модели, а также тел методов и определенных пользователем структур данных. Границы таких фрагментов аннотированы при помощи специальных комментариев, указывающих, к какому структурному элементу модели относится код, написанный пользователем. Алгоритм генерации кода по XML-спецификации на известном языке (в данном случае языке C) является эталонной реализацией, определяющей смысл модели. Алгоритм основан на преобразованиях по шаблонам, так что связь между частями XML-спецификации и сгенерированным кодом легко обнаруживается программистом. Для дополнительного удобства в генерируемый код переносятся комментарии из XML-спецификации.

Алгоритм распараллеливания кода в общем случае заключается в извлечении блоков пользовательского кода из исходной программы; генерации новой программы с синхропримитивами параллелизма по XML-спецификации; включения извлеченных пользовательских блоков в код сгенерированной параллельной программы.

Таким образом, примененный подход является простым, следовательно, надежным; языково-нейтральным, так как не требует лексического и синтаксического анализатора для целевого языка; позволяет точно сформулировать операционную семантику модели программирования в терминах последовательного языка, не прибегая к дополнительным математическим нотациям; программа отлаживается обычным способом.

Алгоритм генерации кода реализован в препроцессоре templet, совместная работа которого с IDE Microsoft Visual Studio организуется, как описано ниже. Пусть код

программы пишется на языке C++, что предусмотрено в текущей реализации. Модулем в данном случае является пара файлов: заголовочный файл с расширением (*.h); файл реализации с расширением (*.cpp). Код этих файлов помещается в общее пространство имен (namespace). Каждому модулю в проекте IDE соответствует файл с XML-спецификацией, который тоже включается в проект (обычно в папку ресурсов приложения). Для файла XML-спецификации в проекте указывается способ его обработки. Перед сборкой проекта (pre-built) такой файл должен быть обработан templet-препроцессором, который генерирует фрагменты кода, соответствующие модели программирования в файлах модуля. Также для файла XML-спецификации указывается редактор. Им может быть как штатный текстовый или XML-редактор, так и специальный графический редактор, визуализирующий модель программирования при помощи графических обозначений рис.1,2.

В процессе итеративной разработки, состоящей из чередующихся шагов редактирования, сборки и запуска программы, пользователь редактирует как XML-спецификацию модуля, так и код внутри модуля, не относящийся к структурным элементам модели. Препроцессор синхронизирует XML-спецификацию и код модуля при каждой сборке. Вопрос контроля ошибок при такой синхронизации решен следующим образом. Сначала проверяется, является ли XML-файл правильно построенным. Далее в процессе разбора SAX-парсером проверяются правила вложенности для тегов, описывающих структурные элементы модели программирования. Атрибуты могут быть пропущены и автоматически заменены значениями по умолчанию. В препроцессоре имеется опция преобразования кода XML-спецификации, восстанавливающая все атрибуты, и форматирующая файл XML-спецификации с учетом вложенности тегов. Никакого контроля при генерации больше не выполняется. Семантика модели (например, соблюдение протокола канала) проверяется во время исполнения путем генерации соответствующих assert-инструкций в коде модуля. Если пользователь сформировал блоки, не соответствующие структурным элементам модели, они выносятся в отдельное место в файле модуля. Если в XML-спецификацию был добавлен новый элемент, для которого не было пользовательского блока, то генерируется пустой блок с комментарием, указывающим на необходимость его определения. Таким образом, основной контроль выполняется компилятором и системой исполнения (в случае assert-вызовов) целевого языка программирования. В текущей реализации это компилятор языка C++.

Описанная схема работы обеспечивает уверенность в правильном функционировании кода параллельной программы, так как весь критический код является открытым для анализа и отладки, а технически сложные части анализа выполняются надежным штатным компилятором.

5. Отображение программ. Расширяемая архитектура системы

Для получения собственно параллельного приложения на заданном API для параллельных вычислений выполняется так называемая процедура отображения на программно-аппаратную архитектуру. Эта процедура делает «распараллеливание» логически отлаженного последовательного кода, ориентируясь на его XML-спецификацию. Виды преобразований, которым может подвергаться исходный код в системе Graphplus templet, показаны на рис. 3.

Кроме собственно распараллеливающих преобразований на рис. 3 показаны также другие виды преобразований для исследования характеристик кода. Имитационное моделирование позволяет, после указания при помощи специальных вызовов длительности исполнения методов, исследовать теоретическое ускорение. Оно предсказывает эффективность проектируемого алгоритма без аналитического расчета. Семантический анализ осуществляет статическую проверку ограничений модели, тем самым безопасно исключает assert-инструкции из кода. Семантический анализ также позволяет выполнить

оптимизацию алгоритма исполнения. Простейшим способом гарантирования надежности программы в терминах модели исполнения является ее автоматическое тестирование с разными значениями затравочного числа генератора случайных чисел. Такое тестирование может производиться с использованием высокопроизводительной кластерной системы или суперкомпьютера.

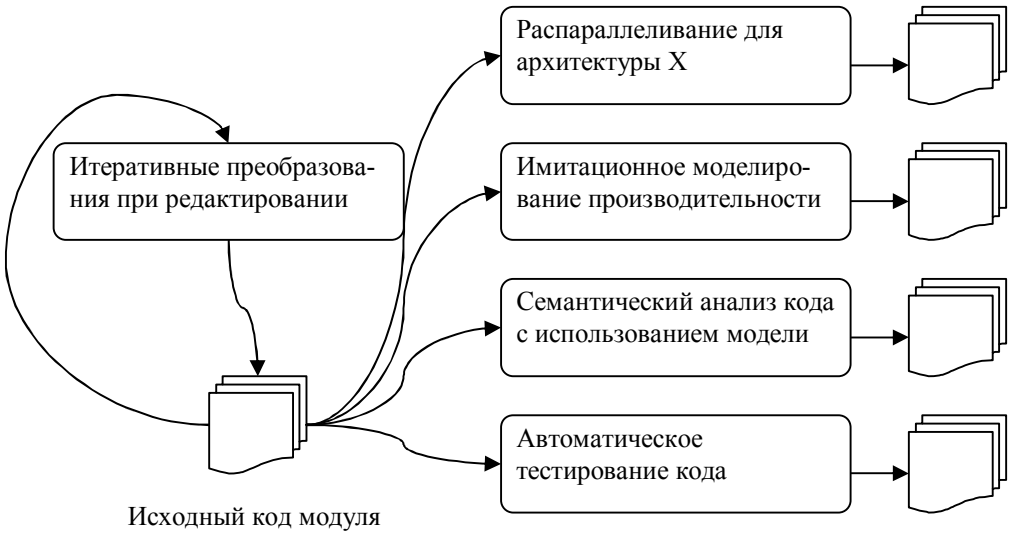

Рис. 3. Виды преобразований исходного кода.

Из примеров, приведенных на рис. 3, видна необходимость расширяемой архитектуры. Имеется еще одна языковая проблема, решаемая с использованием такой архитектуры. Часто оказывается, что выразительных возможностей обычных процессов, каналов и сборок недостаточно для передачи некоторых особенностей параллельных алгоритмов. Например, процессы имеют строго заданное количество портов, что неудобно при описании алгоритмов с коммуникационной топологией «один ко многим». Другой пример – это типовые решения. Применение нестандартной сборки для типовой схемы позволяет использовать заранее определенную коммуникационную топологию процессов и оптимизированный для нее механизм исполнения. Для этого в XML-спецификацию вводятся специальные средства – *templet'*ы и их параметры для процессов, каналов и сборок. Поддержка таких расширений должна быть предусмотрена в архитектуре системы.

Архитектура системы автоматизации параллельного программирования Graphplus templet показана на рис. 4. Система реализована на языке программирования C/C++ без использования системно-зависимого кода. Поэтому она переносится в разные операционные системы путем перекомпиляции. Ядро препроцессора, благодаря особенностям алгоритма проверки соответствия кода модели, является общим для систем программирования разных целевых языков (C++, C#, Java). Ядро выполняет разбор файлов XML-спецификаций; формирует и обеспечивает доступ к описанию модуля; извлекает и формирует пользовательские фрагменты; выполняет другие вспомогательные операции. Для доступа к ним предоставляется специальное API, а само ядро реализовано в виде динамической или статической библиотеки. Генераторы являются исполняемыми программами, использующими ядро и библиотеки расширений для templet'ов. Для вызова библиотек расширений в них имеются API обратного вызова, к которым обращаются генераторы, когда требуется сформировать код нестандартного процесса, канала или сборки.

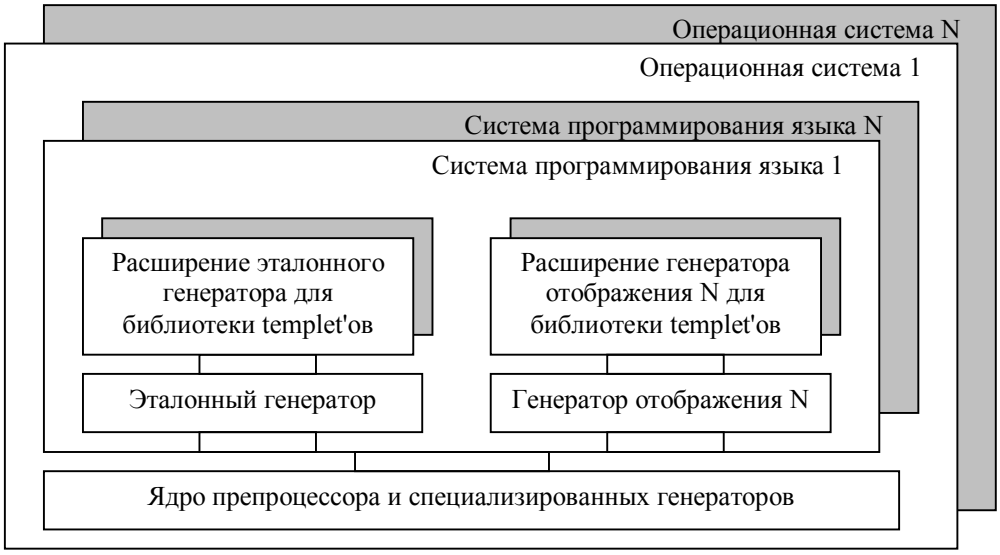

Рис. 4. Архитектура системы программирования Graphplus templet.

Описанная архитектура обеспечивает высокую степень повторного использования кода в системе автоматизации программирования и ее переносимость в различные программно-аппаратные архитектуры. Компоненты, показанные на рис. 4, взаимодействуют с внешним окружением с использованием консольного интерфейса.

6. Пример разработки параллельной программы

Рассмотрим программу, выполняющую проверку тождества $\sin^2 x + \cos^2 x = 1$, где слагаемые вычисляются параллельно. Опишем графически программу в терминах модели программирования системы Graphplus templet. Программа состоит из трех процессов: главного процесса, инициирующего вычисления и вычисляющего сумму; двух идентичных процессов, вычисляющих квадраты тригонометрических функций; канала, описывающего протокол обмена между процессами, рис. 5.

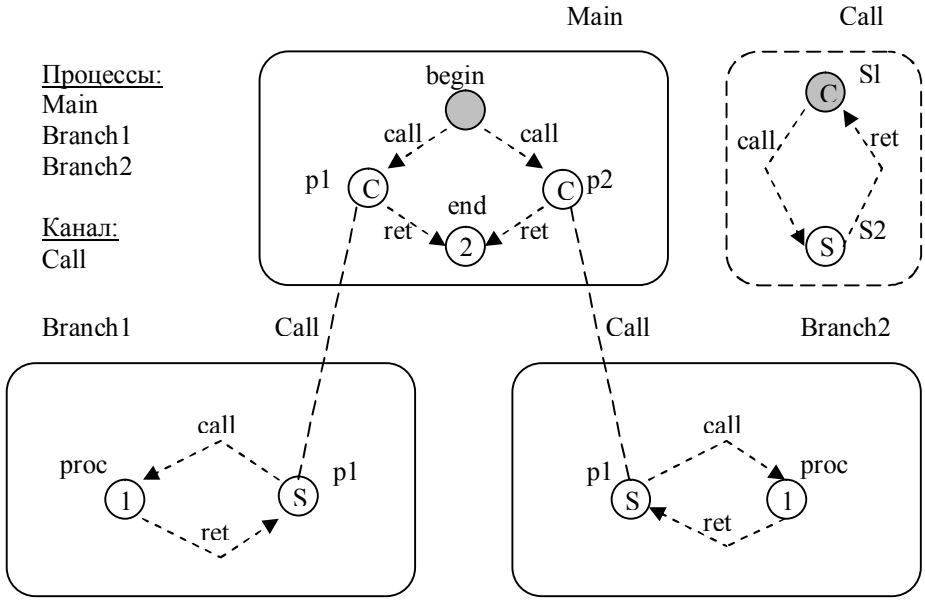

Рис. 5. Модель параллельной программы.

В начальном состоянии поток управления попадает в метод begin процесса Main, который вызывает передачу аргументов *x* в сообщениях call. Процессы Branch, получив сообщения call, вычисляют значения квадратов соответствующих тригонометрических функций и передают их в ответных сообщениях ret. Дождавшись ответов от процессов Branch1 и Branch2, процесс Main вычисляет сумму. Протокол канала Call представляет собой чередование запросов call и ответов ret.

Машиночитаемое представление приведенной модели – это XML-спецификация файла модуля, показанная на рис. 6. В файле перечислены элементы модели и отношения между ними. Формат ориентирован на представление графов модели в виде «вершина – множество исходящих дуг». Формат обеспечивает метод визуализации, так как для каждой вершины и каждого излома дуги предусмотрены атрибуты привязки к координатной плоскости диаграммы. Также из примера видно, что в текстовом представлении контекст метода end можно понять только просмотрев полностью код процесса Main. Графическое представление рис. 5 не обладает таким недостатком. Подготовка XML-спецификации модели штатно выполняется в специализированном графическом редакторе и не вызывает затруднений. В templet препроцессоре имеется поддержка техники ручного редактирования XML-спецификации, когда атрибуты и разметка автоматически генерируются, если указан требуемый тэг. Функция работает аналогично функции auto-completion современных редакторов кода.

```
<module id="Map" templet="GPRT" rem="">
<include file="../rtl/gprt.h" module="GPRT"/>

<channel id="Call" entry="S1" templet="" rem="">
  <state id="S1" type="cli" x="0" y="0" rem="">
         <message id="call" state="S2" x="0" y="0" rem=""/>
  </state>
  <state id="S2" type="srv" x="0" y="0" rem="">
         <message id="ret" state="S1" x="0" y="0" rem=""/>
  </state>
</channel>

<process id="Main" entry="begin" templet="" rem="">
  <port id="p1" channel="Call" module="" type="cli" x="0" y="0" rem="">
         <receive id="ret" method="end" x="0" y="0" rem=""/>
  </port>
  <port id="p2" channel="Call" module="" type="cli" x="0" y="0" rem="">
         <receive id="ret" method="end" x="0" y="0" rem=""/>
  </port>
  <method id="begin" count="1" x="0" y="0" rem="">
         <send id="call" port="p1" x="0" y="0" rem=""/>
         <send id="call" port="p2" x="0" y="0" rem=""/>
  </method>
  <method id="end" count="2" x="0" y="0" rem="">
  </method>
</process>

<process id="Branch1" entry="" templet="" rem="">
  <port id="p1" channel="Call" module="" type="srv" x="0" y="0" rem="">
         <receive id="call" method="proc" x="0" y="0" rem=""/>
  </port>
  <method id="proc" count="1" x="0" y="0" rem="">
         <send id="ret" port="p1" x="0" y="0" rem=""/>
  </method>
</process>

<process id="Branch2" entry="" templet="" rem="">
<!-- аналогично Branch1 -->
</process>

<assemble id="MapAsm" templet="" rem="">
  <process id="Main" module=""/>
  <process id="Branch1" module=""/>
  <process id="Branch2" module=""/>
  <channel id="Call" module=""/>
</assemble>
</module>
```

Рис. 6. XML-спецификация модуля Map.

Программный код процессов и каналов генерируется автоматически. В нем имеются места для вставки объявлений данных и пользовательского кода, что показано на рис. 7, 8. Из листингов видно, как для доступа к данным в сообщениях из методов автоматически определяется список входных и выходных параметров. Кроме того, в методах разрешен доступ к определяемому пользователем состоянию процессов, это имеет место в методах Main::begin и Main::end.

Для отладки логики работы процессов требуется выполнять трассировку кода обработчика входящих сообщений, построенного генератором. На рис. 9 показан такой код для обработчика сообщений процесса Main. Видно, что структура кода сформирована по шаблону из описания процесса рис. 6. Данный код раскрывает семантику формата рис. 6. Соответствие между рис. 5, рис. 6 и кодом рис. 9 очевидно для пользователя системы.

```
class Call:public GPRT::Channel{
public:
  Call(GPRT::Assemble*a);
  ~Call();
public:
  struct call{//<S1>:
/*$GPBS$Call$call*/
  double input;// добавили тип данных в запросе
/*$GPBS$*/
  };
  struct ret{//<S2>:
/*$GPBS$Call$ret*/
  double output;// добавили тип данных в ответе
/*$GPBS$*/
  };
public:
  void _send_call();
  void _send_ret();
public:
  call* _get_call(){return &_mes_call;}
  ret* _get_ret(){return &_mes_ret;}
private:
  Call::call  _mes_call;
  Call::ret  _mes_ret;
// код пропущен
};
```

Рис. 7. Код класса канала Call в заголовочном файле модуля Map.

```
bool Main::begin(/*out*/Call::call*p1,/*out*/Call::call*p2)
{
/*$GPBS$Main$begin*/
  p1->input=in;
  p2->input=in;
  return true;
/*$GPBS$*/
}
bool Main::end(/*in*/Call::ret*p1,/*in*/Call::ret*p2)
{
/*$GPBS$Main$end*/
  out=p1->output+p2->output;
  return true;
/*$GPBS$*/
}
bool Branch1::proc(/*in*/Call::call*p1,/*out*/Call::ret*p2)
{
/*$GPBS$Branch1$proc*/
  p2->output=sin(p1->input)*sin(p1->input);
  return true;
/*$GPBS$*/
}
bool Branch2::proc(/*in*/Call::call*p1,/*out*/Call::ret*p2)
{
/*$GPBS$Branch2$proc*/
  p2->output=cos(p1->input)*cos(p1->input);
  return true;
/*$GPBS$*/
}
```

Рис. 8. Определения пользовательских методов в файле реализации модуля Map.

```
bool Main::_run(int _selector)
{
  for(;;){
         switch(_selector){
                case _port_p1://
                       if(_p1->_c_in_ret()){_selector=_method_end;break;};
                       assert(0);
                       return false;
                case _port_p2://
                       if(_p2->_c_in_ret()){_selector=_method_end;break;};
                       assert(0);
                       return false;
                case _method_begin://
                       assert(_p1->_c_out_call()&&_p2->_c_out_call());
                       if(!begin(_p1->_get_call(),_p2->_get_call()))return false;
                       _p1->_send_call();
                       _p2->_send_call();
                       return true;
                case _method_end://
                       if(++_count_end!=2)return true;
                       _count_end=0;
                       assert(_p1->_c_in_ret()&&_p2->_c_in_ret());
                       if(!end(_p1->_get_ret(),_p2->_get_ret()))return false;
                       return true;
                default: return false;
         }
  }
```

Рис. 9. Код функции обработчика поступающих сообщений процесса Main.

```
int _tmain(int argc, _TCHAR* argv[])
{
  gpInit(new MapAsm());

  int m;//создание главного процесса
  gpProc(&m,MapAsm::Map_Main);
  Map::Main*mprc=(Map::Main*)gpRef(m);

  int b1,b2;//создание процессов для параллельных ветвей
  gpProc(&b1,MapAsm::Map_Branch1);
  gpProc(&b2,MapAsm::Map_Branch2);

  int cli1,cli2,srv1,srv2;//создание каналов
  gpChan(&cli1,&srv1,MapAsm::Map_Call);
  gpChan(&cli2,&srv2,MapAsm::Map_Call);

  gpLink(m,cli1,Main::_port_p1); // связывание процессов каналами
  gpLink(m,cli2,Main::_port_p2);

  gpLink(b1,srv1,Branch1::_port_p1);
  gpLink(b2,srv2,Branch2::_port_p1);

  mprc->in=123;//входные данные
  gpRun();
  std::cout << mprc->out;//выходные данные

  gpFinalize();
  return 0;
}
```

Рис. 10. Код функции main параллельной программы.

Код запуска программы показан на рис. 10. Для выполнения управляющих функций в системе исполнения GraphPlus templet имеется специальный API, функции которого на рис. 10 имеют префикс gp*. При запуске программы выполняется инициализация системы исполнения путем конструирования и передачи в систему объекта сборки. Далее строятся процессы и каналы, выполняется их связывание, запуск вычислений и очистка. На рис. 10 видно, что описание коммуникационной топологии, хотя и требует в настоящее время ручного кодирования, достаточно просто. Допущенные ошибки связывания (например, несоответствие типов) будут обнаружены системой исполнения до запуска вычислительного процесса.

7. Сравнение с известными подходами

В основе представленной системы автоматизации параллельного программирования лежит система визуального программирования GraphPlus [1]. Усовершенствования коснулись модели программирования, в которую была добавлена концепция каналов, используемая в операционной системе Singularity и языке программирования Sing# [2]. Подвергся пересмотру метод определения семантики XML-спецификации. В отличие от работы [3], где семантика модели описана в терминах темпоральной логики, применен метод эталонной реализации [4], упрощающий понимание системы программистом. В язык XML-спецификации введена возможность описания типовых решений (паттернов, шаблонов) [5].

В основе концепции данной системы лежит парадигма языково-ориентированного программирования [6], в рамках которой система может рассматриваться как пример доменно-ориентированного языка для параллельных вычислений. Система предназначена для реализации и использования паттернов параллельного программирования [7] в более гибкой форме, чем в соответствующих каркасных библиотеках. В частности, как в работе [8], единая вычислительная модель служит целям композиции типовых решений. Параллелизм в представленной системе вводится не как расширение языка или собственный язык, а в форме библиотеки времени исполнения [9], однако для обеспечения переносимости определяется лишь эталонная реализация. Система занимает промежуточное положение между языком и каркасной библиотекой, построена на основе препроцессора, и в этом похожа на систему программирования Qt [10]. Однако она не имеет жесткой привязки к языку программирования C++, не выполняет даже лексический разбор языковых конструкций, что значительно упрощает дизайн.

Общеупотребительными являются несколько способов введения семантики параллельного исполнения в язык программирования. Каркасная библиотека, например TBB на языке C++[11], жестко связывает решение с языком программирования. Расширение языка требует написания как минимум лексического анализатора языка, что может оказаться неприемлемо сложным для практического применения, хотя в некоторых случаях удается построить компактные расширения, например T++ [12]. Использование распараллеливающих директив в форме комментариев, например OpenMP [13] или DVM [14], тоже предполагает наличие сложного компилятора, но кроме этого, заставляет пользователя вручную приводить структуру кода в соответствие с моделью программирования этих средств. Использование приема генерации исходного кода вместо его анализа снимает данную проблему.

Семантика кода в представленной системе основана на модели процессов. Данный выбор обусловлен широкими описательными возможностями, императивностью, удобством декомпозиции сложных систем на ее основе. Несмотря на достоинства функциональной модели, и в том числе возможности описания некоторых паттернов параллелизма в ее терминах [15], мы считаем наш подход более универсальным. Использование модели процессов диффузного типа [16] позволяет выполнить ее реализацию базовыми средствами процедурно-ориентированных языков, например, избегать манипулирования со стеком вызовов.

8. Заключение

Эксперименты с созданной нами системой программирования Graphplus templet подтверждают возможность практического применения изложенных в работе решений. Проверка функциональности проведена при проектировании и кодировании наиболее распространенных типовых схем управления вычислениями: «применить ко всем» (MAP); «портфель задач» (TASKBAG); «конвейер / цепь асинхронных процессов» (PIPELINE / CHAIN). Разработана система отображения программ на SMP архитектуру,

использующая синхропримитивы интерфейса Win32 для многопроцессорных и многоядерных компьютеров под управлением операционных систем семейства Microsoft Windows на ядре NT. Примеры иллюстрируют снижение трудоемкости и повышение надежности параллельного и распределенного программирования с использованием системы Graphplus templet.

В комплект программ Graphplus templet входит транслятор модели / препроцессор, модуль с отладочным последовательным кодом системы исполнения, модуль многопоточного исполнения для Win32 API, примеры тестовых приложений и опционный графический редактор файлов XML-спецификации. Система реализована на языке C/C++, использует SAX-парсер Expat 2.0.1 для разбора файлов XML-спецификации. В настоящий момент Graphplus templet поддерживает программирование приложений на языке C++. В прототипах системы также тестировались варианты реализации типовых схем на языках Java и C#. Код системы зарегистрирован Федеральным институтом промышленной собственности (ФИПС). Предполагается использование данной системы автоматизации на суперкомпьютере «Сергей Королев» типа IBM BladeCenter HS22 производительностью 10 TFLOPS, установленном в Самарском государственном аэрокосмическом университете.

Список литературы

1. Востокин С.В. Графическая объектная модель параллельных процессов и ее применение в задачах численного моделирования. Самара: Издательство Самарского научного центра РАН, 2007. 286 с. (http://graphplus.ssau.ru/docs//GraphicalObjectModel.pdf)
2. Galen C. Hunt and James R. Larus, Singularity: Rethinking the Software Stack // ACM SIGOPS Operating Systems Review, vol. 41, no. 2, pp. 37-49, Association for Computing Machinery, Inc., April 2007.
3. Востокин С.В. Спецификация визуальной модели параллельных и распределенных вычислений на основе логики TLA // Труды Четвертой Международной конференции «Параллельные вычисления и задачи управления» PACO'2008. Москва, 27-29 октября 2008 г. М.: Институт проблем управления им. В.А. Трапезникова РАН, 2008, С. 1338-1348.
4. Dalci, Eric; Fong, Elizabeth; Goldfine, Alan. Requirements for GSC-IS Reference Implementations. National Institute of Standards and Technology, Information Technology Laboratory, 2003.
5. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. Спб: Питер, 2003. 368 с.: ил.
6. Дмитриев С. Языково-ориентированное программирование: следующая парадигма // RSDN Magazine, №5. 2005.
7. Шмидт Д., Хьюстон С. Программирование сетевых приложений на C++. Том 2. М.:ООО "Бином-Пресс", 2004. 400 с.: ил.
8. Бергизияров П.К. Программирование на типовых алгоритмических структурах с массивным параллелизмом // Вычислительные методы и программирование. 2001. Т.2, Разд. 2, С.1-16.
9. Страуструп Б. Язык программирования C++, 3-е изд. Спб.: М:. «Невский диалект» - «Издательство БИНОМ», 1999.
10. Земсков Ю.В. Qt 4 на примерах. СПб.: «БХВ-Петербург», 2008.
11. Reinders J. Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media, 2007.
12. С.М. Абрамов, А.И. Адамович, А.В. Инюхин, А.А. Московский, В.А. Роганов, Ю.В. Шевчук Т-система с открытой архитектурой // Суперкомпьютерные системы и их применение SSA'2004: Труды международной научной конференции, 26-28 октября 2004 г., Минск, ОИПИ НАН Беларуси. Минск, 2004. С. 18-22.
13. OpenMP API specification for parallel programming. http://openmp.org. 2010.
14. Крюков В.А., Удовиченко Р.В. Отладка DVM-программ // Программирование. 2001. №3. С. 19-29.
15. Московский А.А., Первин А.Ю., Сергеева Е.О. Первый опыт реализации шаблона параллельного программирования на основе Т-подхода // Программные системы: теория и приложения: тр. Междунар. конф. М.: Наука. Физматлит. 2006. Т. 1.
16. Dijkstra E.W., Scholten C.S. Termination detection for diffusing computations // Information Processing Letters. Augest 1980. No 11(1). P. 1-4.