Министерство образования и науки
Российской Федерации
Федеральное агентство по образованию

Новосибирский Государственный Университет

Факультет информационных технологий

Кафедра параллельных вычислений

Выпускная квалификационная работа магистерская диссертация

Перепелкин Владислав Александрович

RUN-TIME СИСТЕМА ДЛЯ ИСПОЛНЕНИЯ ФРАГМЕНТИРОВАННЫХ ПРОГРАММ

Научный руководитель
д.т.н., профессор
В. Э. Малышкин

Новосибирск 2007

Содержание

ВВЕДЕНИЕ....3
Глава 1. Фрагментированное программирование....12
Технология сборочного программирования....12
1.2 Модель фрагментированного программирования....13
1.3 Фрагментированная программа....19
1.4 Исполнительная система....21
Глава 2. Реализация исполнительной системы....23
2.1 Представление фрагментированной программы....23
2.2 Алгоритм работы диспетчера....24
Характеристика диспетчера....26
Глава 3. Тестирование run-time системы....28
3.1 Фрагментированный алгоритм умножения матриц....28
3.2 Фрагментированный алгоритм LU-разложения....30
Заключение....33
Литература....35
Приложение А. Листинги фрагментированных программ....36
Приложение Б. Описание используемого аппаратного обеспечения....47

ВВЕДЕНИЕ

Параллельное программирование

В настоящее время большие вычислительные мощности, представленные различными параллельными компьютерными системами (многопроцессорные серверы, компьютерные сети, суперкомпьютеры, грид-системы и т.п.), стали доступны широкому кругу пользователей. Это позволяет за приемлемое время решать задачи с большим объемом данных и вычислений, которые раньше невозможно было реализовать на последовательных компьютерах. Но возможности параллельных вычислений непосредственно связаны со сложностями параллельного программирования, которые в свете расширенного круга пользователей встают с особой остротой. Параллельной программе приходится управлять во времени системами, достигающими тысяч процессоров и терабайт оперативной памяти. Рассмотрим основные трудности, возникающие перед программистом.

- Требуется разработать **параллельный алгоритм**, управляющий вычислительной системой («распараллелить» задачу). Последовательные программы не могут, в чистом виде, исполняться параллельно. Корректность параллельного алгоритма отследить трудно, так как приходится иметь дело с несколькими или многими взаимодействующими во времени процессами.
- Необходимо **отобразить этот алгоритм на имеющиеся ресурсы** – распределить вычислительные задачи и данные, обеспечить необходимую синхронизацию в работе и обмен данными, скоординировать совместную работу над исходной задачей. Настройка на имеющиеся ресурсы включает учет особенностей аппаратного и программного обеспечения конкретной системы, таких как объемы дисковой, оперативной и кэш-памяти, тип и производительность вычислительных устройств, задержка и пропускная способность сети, ее топология, имеющиеся программные библиотеки и возможности операционной системы. Для решения этой задачи требуются специализированные знания в области аппаратного обеспечения. В частности, актуальна задача осуществления коммуникаций на фоне вычислений, решение которой требуется для задач, активно использующих как вычисления, так и коммуникации.
- Работа с **распределенными данными** представляет собой нетривиальную задачу. Она требует дополнительных усилий и имеет определенные ограничения. Это необходимо учитывать при разработке параллельного алгоритма. Требуется

реализовать перемещение данных по мере необходимости в конкретных процессорах. Где-то может потребоваться дублирование или поддержка когерентности данных. Каждая из этих задач не имеет универсального решения и должна каждый раз решаться в конкретных условиях задачи. Эффективная реализация может потребовать использования алгоритмов планирования.

- **Распределение работ конкретных процессоров** не всегда известно заранее. Во время исполнения одни процессоры могут простаивать без работы, в то время, как другие полностью нагружены. Такая ситуация требует динамической балансировки нагрузки – перераспределение подзадач в соответствии с нагрузкой на процессоры во время исполнения. Широкий спектр задач численного моделирования без динамической балансировки невозможно решить за приемлемое время.
- Для реализации параллельного алгоритма **требуются языковые средства**, отличные от средств последовательного программирования. В большинстве случаев[1] это расширение какого-либо последовательного языка программирования дополнительными операторами передачи сообщений, синхронизации или групповых операций. Это требует, помимо владения языком последовательного программирования, знания средств параллельного взаимодействия.
- **Отладка параллельных программ** представляет значительно бóльшую сложность, чем отладка последовательных. Это вызвано двумя причинами. Во-первых, требуется отслеживать выполнение программ на многих процессорах. Во-вторых, от запуска к запуску процессы могут отображаться на различные процессоры, коммуникации по аппаратным причинам происходить с различной задержкой и скоростью, операции синхронизации могут срабатывать в разном порядке и т.п. Все это является причиной недетерминизма исполнения параллельной программы. Таким образом, многие даже критические ошибки могут не обнаруживаться длительное время и с трудом поддаются отладке. Такая ситуация известна и в последовательном программировании, но в параллельном масштаб этого явления существенно больше
- **Затруднено использование уже готовых программ** в качестве подпрограмм в других задачах. В настоящее время параллельная программа неразрывна связана со своим программным и аппаратным обеспечением. Эффективность реализации достигается за счет настройки на конкретное оборудование и программное окружение, а в идеале и на конкретные данные. Изменение какой-либо из этих

[1] Менее распространены другие подходы, такие как асинхронное программирования или специализированные языки. Их особенности рассматриваются далее.

составляющих может сильно ухудшить или прекратить работоспособность программы. Как правило, различные средства и системы программирования плохо совместимы. Все это делает практически невозможным использование готовых параллельных программ в качестве подпрограмм в других

Вышеперечисленные трудности, на самом деле, указывают на одну проблему – иметь большие вычислительные мощности – не означает использовать их. На практике даже в хорошо изученном и освоенном последовательном программировании отношение «полезной» производительности процессора к его пиковой производительности превышает 50 процентов лишь для библиотечных функций, разработанных специалистами. Обычные пользовательские вычислительные программы реализуют лишь 10-20 процентов от возможностей процессора. В параллельном программировании эта величина еще значительно меньше. Поэтому вопрос об эффективности реализации занимает одно из главных мест.

Таким образом, программисту приходится тратить значительную часть своих усилий на программирование системных составляющих, не связанных с его задачей. Разработка, отладка и сопровождение параллельных программ требует высокой квалификации системного программирования, которой не обладают, а, главное, и не должны обладать пользователи параллельных компьютеров.

В идеале, пользователь, например физик или биолог, должен иметь возможность с минимальными усилиями запрограммировать алгоритм, получив при этом эффективную параллельную программу, автоматически и динамически настраивающуюся на имеющиеся ресурсы, обеспечивающую полную нагрузку на вычислитель и т.д. Словом, трудности параллельного программирования должны быть скрыты, а нужные качества программы – реализовываться автоматически.

В свете указанных трудностей в параллельном программировании особый интерес представляют различные средства, которые упрощают или автоматизируют разработку, отладку, поддержку и верификацию параллельной программы. Это и определяет актуальность темы работы.

В первую очередь автоматически требуется обеспечивать следующие прикладные свойства параллельной программы:

- Настройка на имеющиеся аппаратные ресурсы
- Динамическая балансировку нагрузки среди имеющихся процессоров
- Осуществление коммуникаций на фоне вычислений

Обзор и анализ средств параллельного программирования

Проанализируем наиболее распространенные системы программирования с позиций автоматизации реализации указанных свойств программы.

Message Passing Interface

Message Passing Interface (MPI) является стандартизованным интерфейсом передачи сообщений. MPI определяет ряд функций, позволяющих передавать и принимать сообщения, осуществлять групповые операции (суммирование, минимизация и т.п.) и синхронизацию. Операции могут выполняться как синхронно, так и асинхронно. Коммуникации осуществляются в рамках т.н. «коммуникатора» - подмножества имеющихся процессоров и связей между ними.

В MPI используется модель передачи сообщений – работает несколько параллельных процессов с раздельной памятью, которые обмениваются сообщениями.

MPI широко используется в большинстве вычислительных кластеров. Он не предоставляет ничего кроме реализации интерфейса и является своего рода «ассемблером» параллельного программирования в том смысле, что реализует лишь низкоуровневые коммуникации, оставляя за пользователем всю работу по управлению работой процессов и коммуникаций.

MPI является слоем виртуализации аппаратного обеспечения и автоматически распределяет процессы по имеющимся процессорам. Это распределение сохраняется во времени, то есть динамической балансировки нагрузки не происходит. Передача сообщений на фоне вычислений поддерживается, но не автоматически. В целом MPI – достаточно низкоуровневое средство и прикладные свойства приходится программировать самостоятельно.

Open Multi-Processing

Open Multi-Processing (OpenMP) – это интерфейс многопоточной обработки общей памяти. Он представляет собой расширение языков Си/Си++ и фортрана в виде директив компилятора и библиотечных функций. Основная идея OpenMP – это автоматическое распараллеливание циклов путем декомпозиции пространства индексов и выполнения цикла в различных легковесных процессах (потоках). OpenMP предоставляет средства для контроля доступа к общим переменным, осуществления групповых (редукционных) операций, а также для синхронизации.

Общий ход выполнения OpenMP-программы выглядит следующим образом. Исполняется основной поток. В точке, где исполнение доходит до параллельного цикла

создается несколько потоков, которые параллельно исполняют цикл, затем происходит синхронизация, и дальше продолжает исполнение только основной поток. Такое ветвление может происходить неоднократно с созданием различного числа потоков.

OpenMP удобно использовать в задачах, где легко производится декомпозиция основных циклов. Менее удобен OpenMP в более сложных задачах, где потоки работают по различной программе и осуществляют нетривиальную синхронизацию.

OpenMP распределяет потоки по имеющимся аппаратным ресурсам. Привязка эта может изменяться в процессе, в том числе автоматически (в основном, средствами работы с потоками операционной системы). Это обеспечивает настройку на имеющиеся ресурсы и динамическую балансировку с точностью до потоков. Коммуникаций как таковых нет, потому что OpenMP работает в общей памяти.

High Performance Fortran

High Performance Fortran (HPF) – это расширение языка Fortran-90, дополняющее его новыми операторами для параллельной работы с массивами. HPF позволяет осуществлять параллельную обработку массивов как в системах с общей, так и распределенной памятью. Программист может задавать рекомендуемый способ разбиения, объявлять «чистые» (от побочных эффектов[1]) процедуры, осуществлять вызовы процедур, использующих собственные библиотеки коммуникаций (таких как MPI).

Таким образом, HPF предоставляет средства для параллельной работы с массивами и совместного использования других средств параллельного программирования. HPF, как таковой, не является универсальным средством параллельного программирования, а лишь позволяет в указанных случаях обрабатывать данные параллельно. Для указанного круга задач HPF обеспечивает необходимые прикладные свойства программ.

Distributed Virtual Machine/Memorsy

Distributed Virtual Machine/Memory (DVM) является расширением последовательного языка (Си, Фортран) и реализуется в форме директив компилятору в комментариях. И для последовательного и для параллельного исполнения, таким образом, используется одна и та же программа. DVM позволяет осуществлять параллельные обработку массивов и выполнение циклов. Задачи и данные при этом распределяются по решетке процессоров. Управление процессом распараллеливается с помощью высокоуровневых спецификаций. По сравнению с HPF DVM предоставляет более

[1] Функция называется функцией без побочных эффектов, если она не использует глобальных ресурсов (переменные, устройства ввода/вывода и т.п.). Функции без побочных эффектов могут безопасно работать параллельно.

широкие возможности по параллельной работе с данными, а также позволяет распараллеливать вычисления (по итерациям цикла). Но принципиально картина остается такой же – язык последовательного программирования, дополненный возможностями параллельной работы для узкого круга задач, и обеспечивающий прикладные свойства программы в нем.

Т-Система

В Т-Системе используются парадигмы функционального программирования для обеспечения динамического распараллеливания программ. Программа строится из «чистых» функций – функций без побочных эффектов. Вызов такой функции порождает параллельный процесс, не блокируя основной. Для корректного возврата результатов работы функции в традиционный язык программирования вводится понятие неготового значения.

Система не требует явного описания организации параллельного исполнения фрагментов, их распределения по узлам, синхронизации, обмена данными. Она сама решает задачу о распределении вычислительной нагрузки. Но параллельный алгоритм должен быть изложен в функциональном стиле, что не всегда удобно.

Т-Система должна обеспечивать прикладные свойства параллельных программ автоматически, но на практике это происходит лишь на определенном классе задач. Это связано со сложностями реализации самой Т-Системы. Дело в том, что параллельная программа, по сути, представлена графом операций, а функции вычисляются в зависимости от готовности данных (dataflow). Эффективное исполнение такого графа является, в общем случае, сложной проблемой отображения этого графа на ресурсы суперкомпьютера во времени.

Норма

Язык Норма – специализированный непроцедурный язык для спецификации задач вычислительного характера. Конструкции языка носят декларативный характер и описывают правила вычисления значений. Система ориентирована на сеточные методы.

Входной язык – язык описания математических соотношений. Отсюда удобство использования системы прикладными программистами: нет необходимости программировать, требуется только описывать задачу. Сохраняется весь внутренний параллелизм алгоритма. Пользователь может пользоваться как библиотечными функциями, такими как sin, cos, sqrt, log и т.п., так и собственными.

Норма отличается высокоуровневым описанием алгоритма в функциональном стиле, а программа генерируется автоматически. Прикладные свойства параллельной программы обеспечиваются, однако область применения ограничивается прямоугольными сеточными вычислениями.

Kernel Lattice Parallelism

Система программирования Kernel Lattice Parallelism (KeLP) предназначена для упрощения реализации научных задач на сетках. Поддерживается динамическая балансировка за счет перемещения атомарных кусков сетки. Данные представляются в виде множества блоков, на которых определяется граф зависимостей. Данные будут размещены по процессорам автоматически в соответствии с заявленными зависимостями. Программа пишется на двух уровнях – KeLP для описания структур данных и параллелизма и Си или Фортран для реализации численных вычислений.

KeLP обладает развитым инструментарием в области сеточных задач. В частности, есть возможность динамической детализации сетки, а динамическая балансировка нагрузки на процессоры осуществляется передачей атомарных фрагментов сетки. Но принципиально картина такая же, как и в языке Норма – KeLP является специализированной системой программирования сеточных задач.

Результаты анализа

Анализ средств параллельного программирования показал, что задачи автоматической настройки на имеющийся ресурс, динамической балансировки нагрузки и реализации коммуникаций на фоне вычислений преимущественно лежит на программисте. Это, в первую очередь, качается MPI – видимо, самого распространенного и универсального решения. Другой распространенный вариант – OpenMP – ограничен системами с общей памятью. Остальные средства параллельного программирования обеспечивают необходимые динамические свойства программ на определенном круге задач.

Цель работы

Требуется разработать средство параллельного программирования, автоматически обеспечивающее необходимые свойства прикладной программы: настройку на аппаратное обеспечение, динамическую балансировку нагрузки и реализацию коммуникаций на фоне вычислений.

В работе используется подход, предложенный в [1] Вальковским В.А. и Малышкиным В. Э., называемый сборочной технологией программирования. Показывается, что его составляющая часть – методика фрагментированного программирования – имеет ряд преимуществ в качестве системы параллельного программирования по сравнению с существующими системами. Спроектирована и реализована система исполнения фрагментированных программ. Произведено ее тестирование на ряде задач математического моделирования, выявлены основные характеристики системы. Сделан вывод о целесообразности использования технологии фрагментированного программирования в задачах численного моделирования.

Глава 1. Фрагментированное программирование

Технология сборочного программирования

Идея сборочного программирования заключается в том, что программист на языке высокого уровня описывает задачу в терминах предметной области (в формальной модели предметной области). Эта модель, как правило, представляет собой множество понятий и правила получения одних величин из других. Программист описывает известные величины и обозначает те, которые бы хотел определить. В соответствии с формальной моделью строится план вычислений – множество операций и допустимый порядок исполнения. Каждая операция вычисляет одни величины из других, и будучи исполненными в допустимом порядке операции плана вычисляют все необходимые значения. Каждая операция поддержана одной или более прикладной подпрограммой, реализующей данную операцию. Эта подпрограмма может быть написана на «обычном» языке программирования, таком как Си/Си++, Фортран и т.п. Затем по составленному плану синтезируется программа, которая представляет собой набор прикладных подпрограмм и правила их сборки и исполнения. Программа синтезируется в соответствии и аппаратным и программным окружением вычислителя, на котором будут производиться вычисления. Затем эта программа исполняется специальной исполнительной системой, которая осуществляет динамическую настройку на вычислительный ресурс, в частности, производит динамическую балансировку нагрузки.

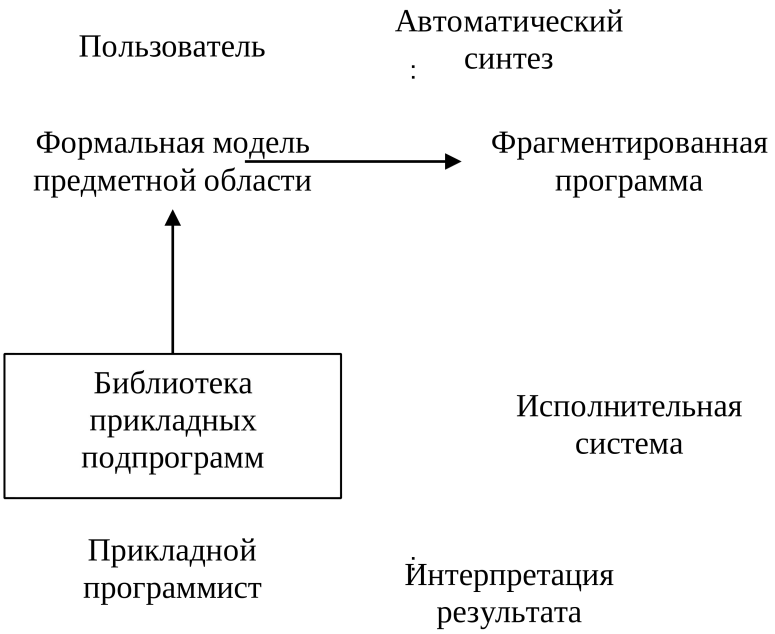


Рис.1.1 Технология сборочного программирования.

Базовой для сборочной технологии является его составная часть – технология фрагментированного программирования. Это инструмент, используемый на стадиях синтеза и исполнения программы. Рассмотрим эту модель более подробно.

1.2 Модель фрагментированного программирования

Фрагментированное программирование – это технология параллельного программирования, частично реализующая сборочный подход. Модель фрагментированного программирования имеет основными элементами фрагменты. Фрагмент является совокупностью процедуры, данных (множество переменных, фрагмент данных) и схемы сборки, в которой он участвует. Кроме этого для каждого фрагмента определены допустимые правила сборки. Каждый такой фрагмент соответствует прикладной поддержке какой-либо операции из формальной модели. Фрагментированная программа представляет собой сборку из множества, вообще говоря, различных фрагментов одним из допустимых способов. Сборка происходит путем отождествления переменных различных фрагментов. Программа, сохраняя фрагментированность, поступает на исполнение вычислительной системе, которая распределяет фрагменты по имеющимся вычислительным ресурсам.

Исполнение программы состоит в том, что последовательно или параллельно осуществляется срабатывание фрагментов – то есть, запуск вычислительной процедуры. Каждый фрагмент из известных значений переменных вычисляет неизвестные значения. Этот процесс происходит до тех пор, пока все необходимые значения переменных не будут вычислены. В процессе работы фрагментированной программы фрагменты могут мигрировать с одного процессора на другой, поддерживая равномерность нагрузки на вычислительный ресурс.

Сборка фрагментов определяет информационные зависимости, которые накладывают соответствующие ограничения на порядок исполнения (т. н. «потоковое управление»): если фрагмент вычисляет значение какой-либо переменной, а затем это значение использует другой фрагмент, то срабатывание первого должно завершиться перед началом срабатывания второго. С целью увеличения эффективности, в каком либо из смыслов, на программу может быть наложено дополнительное управление, ограничивающее допустимый порядок исполнения фрагментов (т. н. «прямое управление»). Прямое управление может быть задано различными способами, например, отношением частичного порядка на множестве фрагментов.

Кроме управления на множестве переменных фрагментов должна быть задана функция отображения на физические ресурсы во времени. Это отображение определяет, каким переменным и в какое время будет выделен ресурс физической памяти для хранения значений. Это отображение может отличаться не только для различных программ, но и для различных запусков одной и той же программы, например, на различных вычислителях.

Рассмотрим необходимость в задании прямого управления и отображения на ресурсы на примере (рис.1.2).

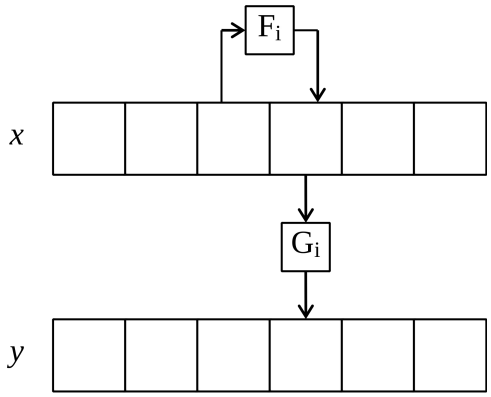


Рис.1.2 Пример фрагментированной программы

Требуется посчитать вектор *y*, являющийся функцией от вектора *x* поэлементно, вычисляется операцией F. Элемент x_i вычисляются из x_{i-1} операцией G. Во фрагментированной программе эти операции реализуются фрагментами F_i и G_i. На практике в целях экономии памяти логично выделить для хранения значений вектора *x* только одну ячейку, последовательно принимающую значения всех x_i. Но в этом случае возникает необходимость срабатывания фрагмента G_i сразу после соответствующего срабатывания F_i. Это задается с помощью прямого управления. Если же памяти достаточно, то можно выделить ресурс для нескольких фрагментов вектора *x* и т.п.

Особенности фрагментированного программирования

Как технология параллельного программирования фрагментированное программирование обладает рядом преимуществ. В первую очередь, это возможность конструировать параллельные программы из фрагментов, каждый из которых содержит обычную последовательную процедуру. В качестве такой процедуры можно использовать библиотечную функцию. При этом программисту не нужно вручную программировать передачу сообщений или динамическую балансировку нагрузки – эти и другие системные задачи берет на себя исполнительная система. Фрагментированная программа или подпрограмма может работать при любом начальном распределении фрагментов данных. Исполнение фрагментированной программы может осуществляться как на последовательном вычислителе с небольшой производительностью и объемом памяти, так

и на мощном кластере или грид-системе. Другие особенности фрагментированного программирования:

- Алгоритм задачи изначально представлен в параллельном виде. Это означает, что автоматическое распараллеливание (которое, в общем случае, принципиально нереализуемо эффективно) не требуется. Степень непроцедурности[1] в дальнейшем будет уменьшаться, но изначально параллелизм уже заложен в фрагментированном представлении алгоритма.
- Возможно использование множества ранее созданных фрагментов в новых задачах. Это делает возможным накопление эффективно реализованных и отлаженных фрагментов с целью их последующего использования. Кроме этого, возможно использовать имеющуюся схему фрагментированной программы, заменив фрагменты или процедуры фрагментов. Это позволяет использовать отлаженные схемы программ для осуществления аналогичных вычислений. Таким образом, накопление параллельных решений может осуществляться на двух уровнях – уровне готовых фрагментов и уровне готовых схем фрагментированных программ.
- Пользователь, как правило, составляет программу из уже готовых фрагментов. Это означает, что от пользователя не требуется знание какого-либо другого языка программирования. Он просто описывает алгоритм в терминах, близких к терминам самой задачи, сохраняя его параллелизм. Если же пользователю понадобится создать новый фрагмент, то для его реализации достаточно знание одного из языков последовательного программирования, либо он может отправить запрос прикладному программисту, и тот разработает необходимую процедуру.
- Фрагментированность программы сохраняется на стадии исполнения. Фрагмент является относительно автономным элементом, который можно перемещать на менее загруженный процессор. Так как схема сборки программы известна, то балансировку можно планировать заранее, сводя простои к минимуму.
- Размер фрагмента можно изменять таким образом, чтобы а) фрагмент был достаточно маленьким, чтобы размещаться в кэше и б) фрагмент был достаточно большим, чтобы основное время занимали вычисления, а не накладные расходы на управление фрагментами.
- Автоматически достигается реализация коммуникаций на фоне вычислений. Дело в том, что если фрагментов достаточно много, то часть находится в ожидании

[1] Непроцедурность понимается в смысле [2]. Непроцедурность представления алгоритма характеризует содержащийся в нем параллелизм.

данных, а часть – в ожидании исполнения, поэтому в одно и то же время выполняются и коммуникации, и вычисления.

- Фрагмент ограничивает возможные способы сборки. Сборка фрагментов допустима, если она корректна в смысле предметной области (два фрагмента нельзя соединить по переменным одного типа, если они имеет разный смысл). Таким образом, процесс сборки сохраняет корректность. Если же пользователь хочет осуществить сборку не по правилам, то он должен явно это указать.
- Отладка фрагментированной программы представляется значительно более простой. Она сводится к двум шагам: а) последовательная отладка процедур фрагментов и б) анализ корректности схемы сборки. Отладка последовательных программ значительно проще сама по себе и существует множество эффективных способов отладки. Анализ же корректности сборки может быть осуществлен математическими методами, так как модель оказывается достаточно простой – граф data-flow, где каждый (атомарный) фрагмент вырабатывает одни значения и потребляет другие. Таким образом, отладка параллельной программы сводится к двум хорошо исследованным и развитым областям.

Технология фрагментированного программирования имеет ряд недостатков и трудностей:

- Требуется создание достаточно сложной автоматизированной исполнительной системы. Она должна обеспечивать управление запуском и распределением фрагментов, осуществлять динамическую балансировку нагрузки на процессоры, реализовывать коммуникации на фоне вычислений, настраиваться на имеющееся аппаратное обеспечение и т.п.
- Требуется разработка фрагментированных алгоритмов.
- Возникают дополнительные расходы памяти, связанные с фрагментацией данных и хранением системной информации о фрагментах.
- Возникают дополнительные вычисления, связанные с обработкой системной информации, управлением исполнением фрагментов и передачей данных.

Эти трудности представляются приемлемыми по сравнению с конечным результатом, то есть возможности программировать на высоком уровне с автоматическим обеспечением динамических свойств программы.

Формализуем модель фрагментированного программирования.

Основным элементом является фрагмент f=<V, P, A>, где V – это переменные (входные и выходные), P – операция, вычисляющая значения выходных переменных из

входных, и A – допустимые способы сборки данного фрагмента с другими. V = <$v_1 \ldots v_k$>. A={a|a=<f_{i1},v_{i2},f_{j1},v_{j2}>, v_{i2} in(f_{i1}), vj_2 out(f_{j1})}, где in(f) и out(f) – входные и выходные переменные фрагмента f соответственно.

Фрагментированная программа FP=<F, C, M>, где F=<$f_1 \ldots f_n$>, C – прямое управление, M задает отображение фрагментированной программы на ресурсы во времени – процессоры, память и т.п. Срабатыванием фрагмента f называется вычисление значений out(f) из значений in(f) с помощью операции P. Прямое управление C задает способ исполнения фрагментированной программы – информационные зависимости, ограничения на допустимый порядок исполнения, приоритеты и т.п.

Исполнением фрагментированной программы FP называется процесс срабатывания фрагментов в соответствии с прямым и потоковым управлением до тех пор, пока не сработают все фрагменты, либо значения всех выходных переменных не будут вычислены. Исполняться фрагментированная программа может в общем случае различными способами.

Это модель единственного присваивания и единственного срабатывания каждого фрагмента. На практике из соображений реализуемости будет использована другая модель, более низкоуровневая и более близкая к программированию. Именно, каждый фрагмент может срабатывать более одного раза и переменные понимаются в программном, а не математическом смысле – то есть как ячейки памяти. На время срабатывания фрагмента его переменные блокируются для доступа со стороны других фрагментов. Блокировка может быть на чтение (не эксклюзивная) или на запись (эксклюзивная).

Рассмотрим пример – умножение двух матриц по известной формуле $C_{i,j} = \sum_{k=1}^{N} A_{i,k} \times B_{k,j}$. Последовательная реализация данного алгоритма на псевдоязыке может выглядеть следующим образом:

```
type Matrix : array[N,N] of real
var A, B, C : Matrix
// init A and B here
for i=1..N
        for j=1..N
                C[i,j] = 0
                for k=1..N
                        C[i,j] = C[i,j] + A[i,k]*B[k,j]
                end for
        end for
```

```
end for
```

Фрагментированная программа бы выглядела примерно следующим образом (опишем ее на неформальном псевдоязыке):

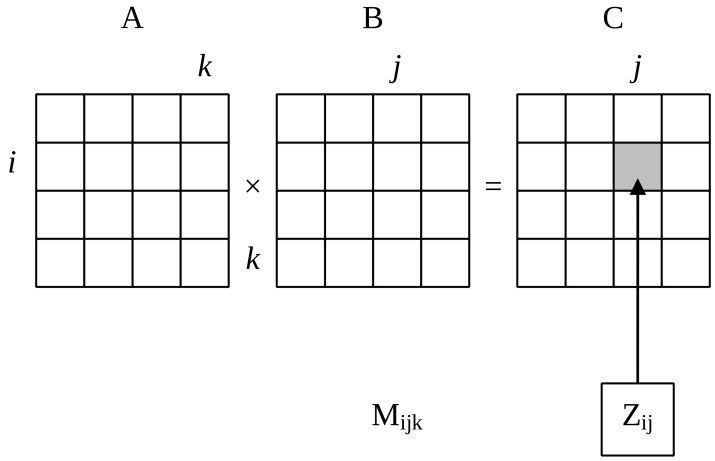


Рис.1.3 Схема фрагментированной программы

Формальные фрагменты.

Фрагмент ZeroMatrixFragment, обнуляет все ячейки матрицы M. Переменные:

M : writeonly Matrix;

Фрагмент MulAddMatrixFragment, умножает матрицу X на Y и результат суммирует с матрицей Z. Результат записывается в матрицу Z

X : readonly Matrix

Y : readonly Matrix

Z : readwrite Matrix

Программа.

Ресурсы: A, B, C : array[N,N] of Matrix; (данные матриц A и B считаем инициализированными)

Фрагменты:

Z : array[N,N] of ZeroMatrixFragment;

M : array[N,N,N] of MulAndAddMatrixFragment;

Отображение на ресурсы и сборка:

Z[i,j].M ≡ C[i,j]

M[i,j,k].X ≡ A[i,k]

M[i,j,k].Y ≡ B[k,j]

M[i,j,k].Z ≡ C[i,j]

Управление:

Z[i,j] < M[i,j,k]

$M[i,j,k_1]$ <> $M[i,j,k_2]$ если $k_1 \neq k_2$ (знак «<>» означает несовместность выполнения двух фрагментов).

Данная фрагментированная программа компилируется в специальное внутреннее представление и передается на запуск исполнительной системе.

1.3 Фрагментированная программа

Внутреннее представление, в зависимости от его спецификации или реализации диспетчера, может отличаться. Оно должно содержать следующую информацию:

- Множество фрагментов
- Управление (прямое и потоковое)
- Описание ресурсов (памяти)
- Отображение данных фрагментов на ресурсы

Опишем структуру фрагментированной программы. Из соображений удобства представления и реализации множество фрагментов программы разбивается на группы. Каждая группа может содержать фрагменты или другие группы. Вся программа образует дерево, узлами которого являются группы, а листьями – дерево. Корень дерева образует *корневую группу*, содержащую все фрагменты программы. Внутри каждой группы может быть определено отношение частичного порядка, задающее управление в этой группе.

В любой конкретный момент времени каждый элемент фрагментированной программы (фрагмент или группа) находится в одном из четырех состояний: «не готов» - не удовлетворены требования управления, «готов» - удовлетворены требования управления, «исполняется» – элемент находится в состоянии вычислений и «завершен» – вычисления завершились. Кроме этого для каждого элемента определено, сколько раз он должен сработать и сколько раз он уже сработал.

Исполнение фрагментированной программы определяется рекурсивно. Из «готовых» элементов корневой группы выбирается один с наибольшим приоритетом. Если он оказывается фрагментом, то запускаются его вычисления. После этого фрагмент в зависимости от числа завершенных срабатываний либо переходит в состояние «завершен», либо – в состояние «готов» с инкрементом счетчика срабатываний. Выбор и срабатывание элементов продолжается до тех пор, пока все они не перейдут в состояние «завершен».

Если выбранный элемент оказался группой, то процедура срабатывания применяется к ней. Когда все элементы перейдут в состояние «завершен» группа так же переходит в состояние «завершен» или «готов» с инкрементом счетчика срабатываний и

сбросом состояния всех дочерних элементов в исходное, то есть со сброшенным счетчиком срабатываний и переходом в состояние «готов»/«не готов».

Группы бывают двух видов – инклюзивные и эксклюзивные. Инклюзивные группы допускают одновременное исполнение своих элементов, а эксклюзивные – нет.

Рассмотрим пример – преобразование фрагментированного вектора *x* в вектор *y* с использованием единственного буфера *b*.

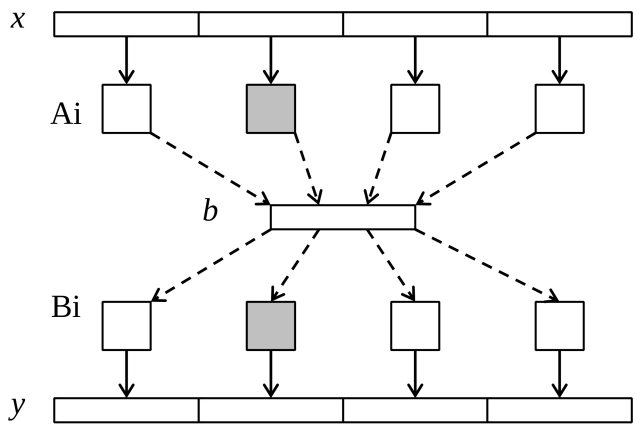


Рис.1.4 Схема фрагментированного алгоритма

Имеется множество фрагментов A_i и B_i. Срабатывание фрагментов Ai может происходить в любом порядке, но не одновременно, причем сразу после выполнения фрагмента Ai должен сработать соответствующий фрагмент Bi, иначе значение в буфере *b* затрется другими данными. Такое управление задается следующим образом. Корневая группа является эксклюзивной и содержит подгруппы G_i, каждая из которых содержит фрагменты A_i и B_i соответственно. Внутри группы G_i задан порядок $A_i < B_i$ (тип групп G_i в данном примере не важен).

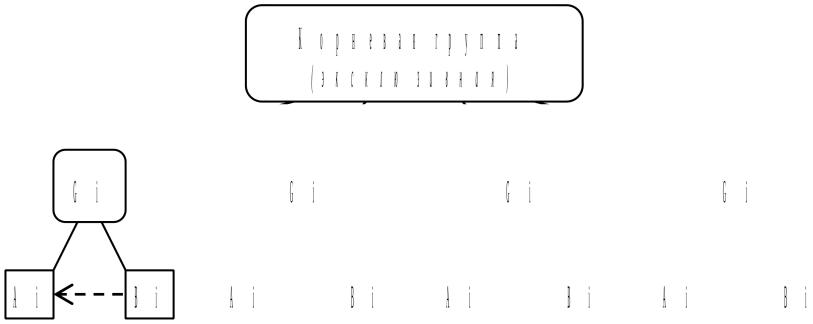


Рис.1.5 Структура фрагментированной программы

Если число повторов фрагмента превышает единицу, то вызов процедуры фрагмента осуществляется необходимое число раз. После этого фрагмент считается исполненным. Если количество повторов для группы превышает единицу, то исполнение группы так же осуществляется несколько раз. Перед каждым повтором состояния ее элементов сбрасываются в исходные.

1.4 Исполнительная система

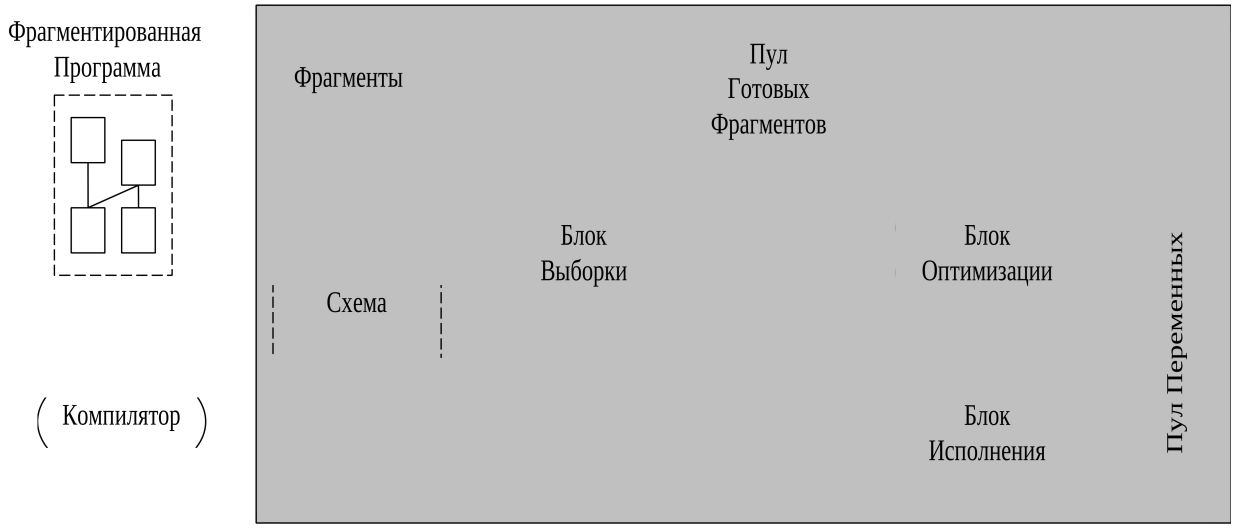


Рис.1.6 Схема исполнительной системы.

Фрагментированная программа записывается на языке высокого уровня (языке фрагментированного программирования). Компилятор преобразовывает фрагментированную программу во внутреннее представление, которое зависит от конкретной реализации диспетчера. Это представление содержит описание самих фрагментов, отношении соседства, переменные, по которым произошла сборка и схему самой сборки. Кроме этого программой определено управление, ограничивающее допустимый порядок исполнения. Это представление отдается на исполнение диспетчеру фрагментов.

Диспетчер состоит из блоков выборки, оптимизации и исполнения, которые работают с внутренним представлением, пулом (хранилищем) готовых фрагментов и менеджером переменных.

Блок выборки в соответствии с готовностью данных и управлением выбирает фрагменты в пул готовых фрагментов. Затем блок оптимизации выбирает в зависимости от аппаратного обеспечения и ресурсов и в соответствии со стратегией оптимизации определенные фрагменты из пула готовых, назначает ресурсы и отдает блоку исполнения. Блок исполнения запускает вычислительную процедуру фрагмента, которая использует

менеджер переменных для получения исходных значений и сохранения вычисленных. По завершению фрагмента уведомление поступает в блок выборки, который просматривает соседей завершенного фрагмента на предмет готовности и при необходимости добавляет их в пул готовых фрагментов.

Пул переменных обеспечивает синхронизацию при доступе к переменным. Перед исполнением фрагмента его переменные захватываются на чтение или запись и освобождаются по завершении работы фрагмента.

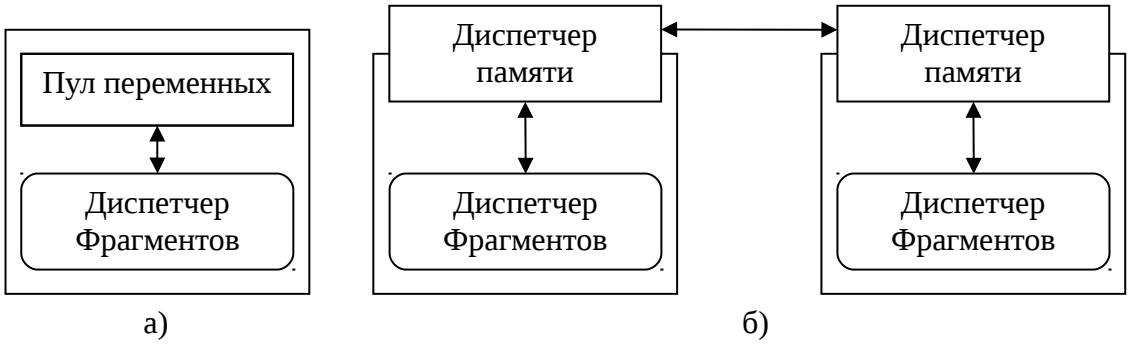


Рис.1.7 Структура диспетчера в узле (а) и в мультикомпьютере (б)

Несколько таких диспетчеров может быть объединено для работы над одной задачей, например, для запуска в мультикомпьютере[1]. В этом случае для управления переменными менеджер переменных становится распределенным. и занимается хранением и перераспределением переменных по узлам мультикомпьютера. Для передачи данных между узлами может использоваться один из интерфейсов передачи сообщений, например MPI. Перед диспетчером памяти стоит задача принятия оптимизационных решений по управлению данными, в частности – собирать данные в один узел перед срабатыванием очередного фрагмента, дублирование данных, нужных для фрагментов в нескольких узлах, освобождение ненужных более ресурсов и т.п.

Каждый блок может быть реализован многопоточно (в особенности, блок вычислений). Это позволит занять имеющиеся процессоры и ядра мультипроцессора.

[1] Мультикомпьютер – параллельная вычислительная система, память которой распределена по ее узлам.

Глава 2. Реализация исполнительной системы

В рамках дипломной работы была реализована система исполнения фрагментированных программ в виде диспетчера фрагментов над общей памятью (для мультипроцессора[1]). Диспетчер был назван RFES (Run-time Fragment Execution System). Реализована несколько упрощенная модель диспетчера. Основное предназначение этой run-time системы – осуществлять исполнение фрагментированных подпрограмм в контексте обычной последовательной программы.

В реализации фрагментированная программа представляется деревом объектов C++. Группы (инклюзивные и эксклюзивные) представлены объектами системного класса Group. Фрагменты представляются объектами классов-наследников системного класса Fragment. Класс-наследник переопределяет метод Procedure, реализующий вычислительную функцию. Процесс создания объектов, представляющих фрагменты и группы является, фактически, построением дерева фрагментированной программы. При этом возможно задавать частичный порядок управления для любой из групп, а также определять приоритеты фрагментов и групп.

Run-time система RFES представлена экземпляром класса Rfes, реализующим диспетчер и предоставляющим доступ к корневой группе и функции запуска. Рассмотрим отдельные части программы более подробно

2.1 Представление фрагментированной программы

В реализации фрагментированная программа представляется в виде дерева объектов, соответствующих группам и фрагментам.

Для представления групп создаются объекты класса Group. Для представления фрагментов создаются объекты классов, дочерних к Fragment. Оба этих класса наследуют класс Item, который соответствует произвольному элементу дерева фрагментированной программы. Он объединяет общую часть групп и фрагментов, так как их обработка во многом одинакова. Класс Item содержит атрибуты Parent – родительская группа и Priority – приоритет элемента. Эти атрибуты доступны только для чтения и задаются при создании фрагмента или группы. Кроме этого Item содержит множество Required, элементами которого являются все элементы, от которых зависит данный. Это множество представляет ту часть частичного порядка, которая относится к данному элементу. Для добавления зависимости определен метод AddRequirement. Определены атрибуты Repeats и Iteration, представляющие общее количество повторов элемента и текущую итерацию. По

[1] Мультипроцессор – параллельный вычислитель с общей памятью.

умолчанию Repeats равен единице, но может быть установлен в любое другое положительное значение. Кроме этого, объявлены булевы методы IsReady, IsRunning, IsDone и IsFragment для определения состояния элемента и его типа. Эти методы реализованы в классах Group и Fragment соответствующим образом. Аналогично объявлены методы IterationDone и Reset – обработчики событий завершения выполнения группы или фрагмента и их сброса в исходное состояние (в начале и перед повторами).

Класс Group наследует класс Item и реализует объявленные в нем функции. Кроме этого, в классе Group определено множество Items, содержащие дочерние элементы группы, и атрибут Excluding, определяющий тип группы. Тип указывается при создании группы и далее доступен только для чтения.

Класс Fragment так же наследует класс Item и реализует объявленные в нем функции. Дополнительно Fragment содержит атрибуты running и done, описывающие текущее состояние фрагмента. Кроме этого объявлена функция Procedure. Эта функция реализует вычислительную процедуру фрагмента, ее переопределяют классы-наследники.

Для представления программы необходимо создать иерархию групп путем создания объектов-групп с нужными аргументами. Затем в эти группы добавляются объекты-фрагменты. Далее над этими объектами задается отношение порядка. После этого фрагментированная программа готова к запуску.

2.2 Алгоритм работы диспетчера

В настоящей реализации интерфейс диспетчера представлен одной функцией Run, осуществляющей запуск фрагментированной программы на исполнение. Кроме этого пользователю открыт доступ к группе SuperGroup для добавления дочерних групп и фрагментов.

Внутренняя структура диспетчера

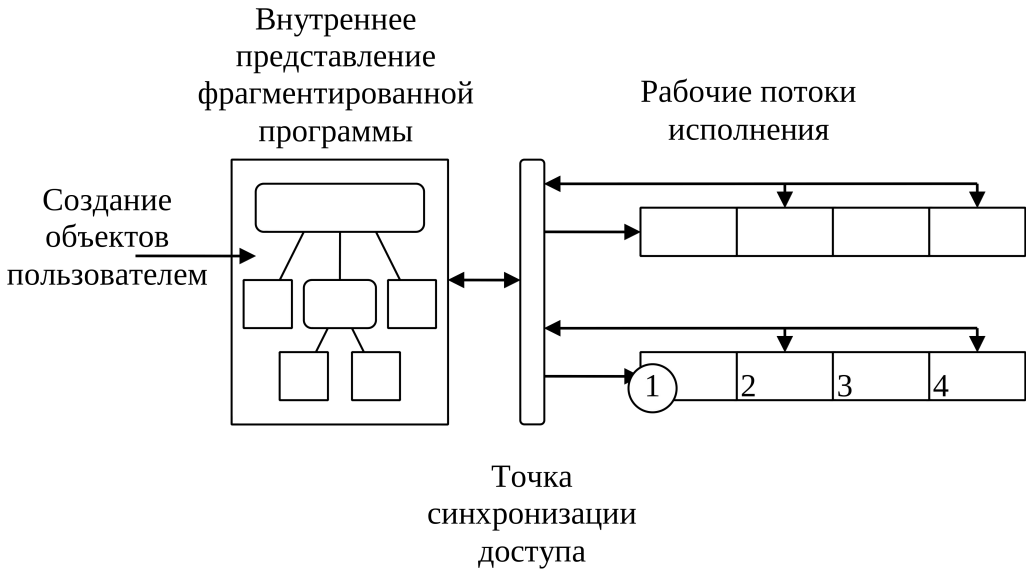


Рис.2.1 Структурная схема диспетчера

Получив команду на исполнение фрагментированной программы диспетчер создает несколько рабочих потоков. Каждый поток работает по одной и той же схеме:

- Выбирается наиболее приоритетный фрагмент из всех, готовых к исполнению.
- Обновляется состояние этого фрагмента во внутреннем представлении программы. Фрагмент помечается как «исполняющийся».
- Осуществляется вызов вычислительной процедуры фрагмента.
- Фрагмент помечается как «завершенный». При этом происходит проверка дерева программы – если фрагмент или одна из родительских групп должна исполниться более одного раза, и на текущий момент исполнение завершено, то происходит сброс состояния с инкрементом текущей операции.

Поток завершает работу, когда вся программа исполнена. При завершении работы всех потоков диспетчер завершает свою работу.

В реализации run-time системы использовались классы Item, Group, Fragment и Rfes, а также вспомогательный класс ItemHasher. Класс Item представляет обобщенный элемент дерева фрагментированной программы – группу или фрагмент. Структура представления фрагментированной программы уже описывалась выше, поэтому остановимся только на внутренних реализационных особенностях. Каждый объект класса Item или дочернего от него имеет мьютекс для синхронизации работы с аттрибутами объекта. Объект хранит приоритет элемента и указатель на его родительскую группу. Для хранения зависимостей (управления) элемента используется класс set стандартной библиотеки шаблонов C++. Хранятся значения количества повторов и номера текущей итерации. При сбросе состояния элемента счетчик итераций будет установлен в ноль.

Функция AddRequirement просто добавляет элемент в множество зависимостей, но перед этим проверяет его наличие в этом множестве. Если элемент уже добавлен, то повторно он не добавляется. Объявляются виртуальные функции проверки состояния и обработчики событий сброса состояния и завершения исполнения. В дочерних классах эти функции будут определены. Функции проверки состояния определены рекурсивно. Функция IsReady рекурсивно вызывается для родительской группы, остальные – для дочерних элементов. Логика состояний фрагментов и групп следующая:

- Фрагмент или группа готовы, если готова родительская группа, и при этом если она исключающая, то не находится в работающем состоянии. Кроме этого должны быть удовлетворены зависимости по управлению.
- Фрагмент работает или завершен, если соответствующие атрибуты класса имеют истинные значения.
- Группа работает, если работает хотя бы один из ее элементов
- Группа завершена, если завершены все ее элементы

Функция сброса состояния так же реализована рекурсивно. При сбросе состояния группы сбрасывается состояние всех ее элементов. Функция завершения итерации работает по следующей схеме. Если это была не последняя итерация, то состояние текущего элемента сбрасывается. Иначе проверяется состояние готовности родительской группы, и если значение истинно, то вызывается обработчик окончания итерации для родительской группы.

Реализация приоритетов использует следующую идею. Класс set хранит множество в упорядоченной форме. Класс ItemHasher переопределяет функцию сравнения элементов в соответствии с их приоритетами. В связи с этим наиболее приоритетные элементы оказываются в начале списка, поэтому проверка состояний проходит подряд с начала списка. На деле реализация усложняется лишь тем, что обход проходит не линейно по множеству, а рекурсивно с углублением в подгруппы. Но идея остается прежней.

Характеристика диспетчера

Основным узким местом диспетчера является централизованная работа с фрагментированной программой. Это непринципиально, если время обработки этих структур данных намного меньше, чем время вычислений, но сказывается, когда они сопоставимы. Количество фрагментов, с которыми свободно может оперировать диспетчер экспериментально проверено в главе 3.

Другим узким местом являются алгоритмы проверки и обновления состояния элементов фрагментированной программы, а также поиска готовых фрагментов. В реализации выбраны алгоритмы, довольно простые в реализации. С одной стороны это позволит при тестировании определить оценку накладных расходов диспетчера, а с другой стороны – ввиду простоты легко видеть, что результаты работы этих функций корректны. Система спроектирована так, что в дальнейшем эти алгоритмы можно с легкостью заменить на более эффективные. Состояния элементов следует кэшировать, а при изменении состояния какого-либо фрагмента или группы обновлять состояния его непосредственных соседей по группе, дочерних и родительских элементов. Функцию поиска можно оптимизировать множеством различных способов, используемых, например, в системах управления базами данных.

Кроме этого не реализована возможность внутренней оптимизации фрагментированной программы во время исполнения. Такая оптимизация является частью технологии фрагментированного программирования. Однако, для такой оптимизации оставлены соответствующие места в схеме диспетчера.

Перечисленные недостатки не являются недостатками технологии фрагментированного программирования как такового, и присутствуют лишь в данной конкретной реализации. Они не являются существенными в отношении тех задач, которые ставились перед этой реализацией. Устранение этих и других недостатков является частью естественного развития исполнительной системы.

Глава 3. Тестирование run-time системы

Было проведено тестирование диспетчера на задачах умножения матрицы на матрицу и LU-разложения. Фрагментированные версии этих алгоритмов были предложены Чаюк К.Г. [3, 4]

3.1 Фрагментированный алгоритм умножения матриц

Был реализован фрагментированный алгоритм умножения матриц $(C = A \times B)$. Матрицы представлены в виде композиции $N \times N$ фрагментов данных, каждый из которых является подматрицей размера $M \times M$.

$C_{i,j} = \sum_{k=1}^{N} A_{i,k} \times B_{k,j}$, где $C_{i,j}, A_{i,k}, B_{k,j}$ – соответствующие фрагменты данных, а умножение понимается в смысле умножения матриц.

Фрагментированная программа состоит из фрагментов двух видов: Zero и MulAdd. Zero обнуляет фрагмент матрицы, а MulAdd умножает два фрагмента и прибавляет к третьему (результат сохраняется в третьем). Группы ${}^{x}G_{i,j}$ являются эксклюзивными

$Z_{i,j} : Zero(C_{i,j}); i, j = 1, \ldots, n$,

$F_{i,j,k} : MulAdd(A_{i,k}, B_{k,j}, C_{i,j}); i, j, k = 1, \ldots, n$

$SuperGroup = \{Z_{i,j}, {}^{x}G_{i,j} \mid i, j = 1, \ldots, n\}$

${}^{x}G_{i,j} = \{F_{i,j,k} \mid k = 1, \ldots, n\}; i, j = 1, \ldots, n$

$Z_{i,j} < {}^{x}G_{i,j}$

Фрагменты $Z_{i,j}$ обнуляют фрагмент матрицы $A_{i,j}$, после чего в произвольном порядке срабатывают фрагменты $F_{i,j,k}$, добавляющие частичные суммы к соответствующему фрагменту матрицы $C_{i,j}$.

Исходный код фрагментированной программы приведен в приложении А Общее количество фрагментов составляет $N^2 + N^3$, где N – число фрагментов данных вдоль одной из осей.

В фрагментированном алгоритме сразу большое число фрагментов готово к исполнению, поэтому распараллеливание задачи должно быть близким к 100 процентам. Отклонение от этой величины может быть вызвано аппаратными характеристиками и накладными расходами на работу диспетчера. Тестирование проводилось на четырех-процессорной системе архитектуры Intel® Itanium®2 «smp4x64» (см. Приложение Б).

Параметрами теста были размер фрагмента данных и количество потоков исполнения. Общий размер данных при этом оставался неизменным: 960×960.

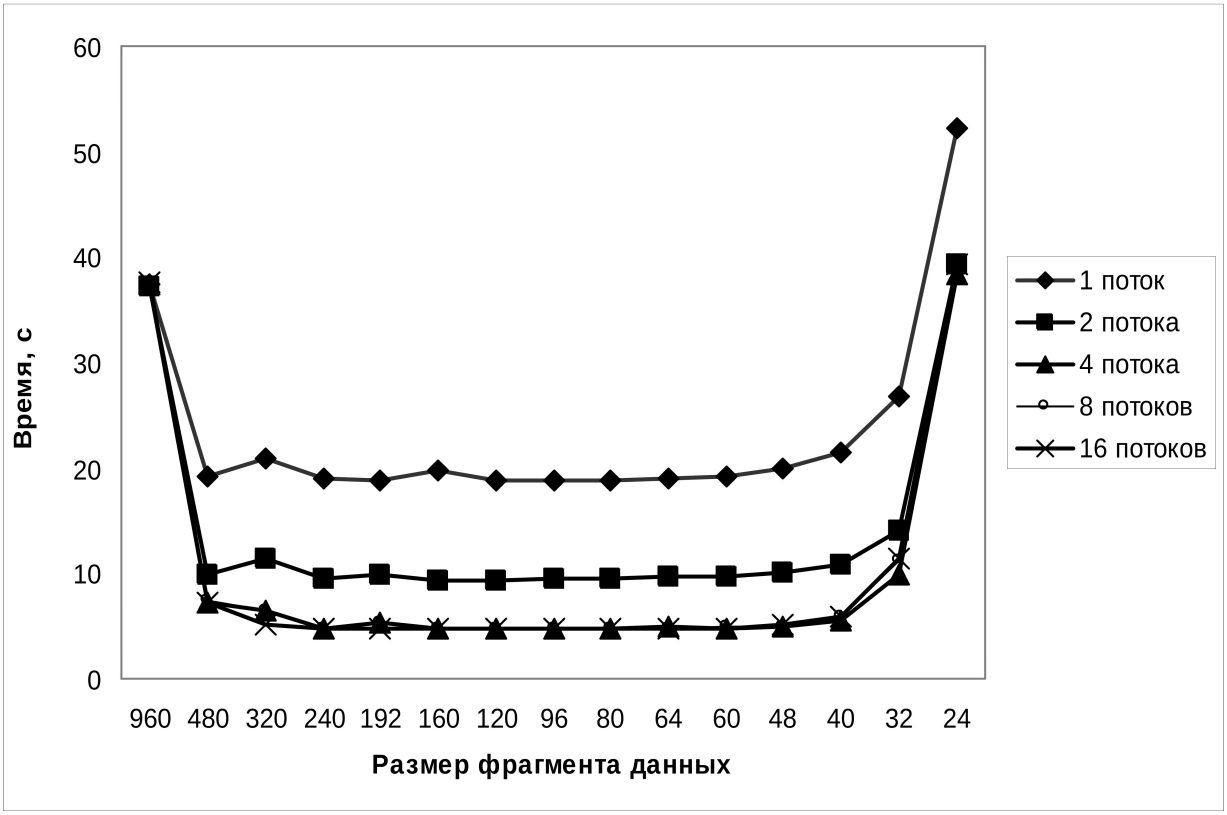


Рис.3.1 Временные характеристики фрагментированной программы умножения матриц.

Из результатов видно, что даже при последовательном (однопоточном) исполнении наблюдается ускорение вычислений. Это связано с тем, что вычисления производятся над различными кусками данных, каждый из которых умещается в быстрой кэш-памяти. В случае последовательного (не фрагментированного) исполнения данные в кэше не умещаются и задействована более медленная оперативная память. Начиная с размера фрагмента 480×480 возникает 4 независимых потока управления ($Z_{ij}, F_{i,j,k}$), что позволяет использовать имеющиеся 4 процессора на 100 процентов. На графике видно, что ускорение на двух потоках двукратное, а на четырех и более – четырехкратное. Влияние run-time системы не выходит за рамки 5 процентов отметки для фрагментов размерами от 240×240 до 60×60 элементов.

3.2 Фрагментированный алгоритм LU-разложения

Алгоритм LU-разложения заключается в том, что матрица A раскладывается на произведение $L \times U$, где L - нижнетреугольная матрица с единичной диагональю, а U – верхнетреугольная матрица.

Алгоритм LU-разложения следующий:

1) $u_{1j} = a_{1j},\ j = 1, \ldots, n$

2) $l_{j1} = \dfrac{a_{j1}}{u_{11}},\ j = 1, \ldots, n$

для $j = 2, \ldots, n$

1) $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj},\ j = i, \ldots, n$

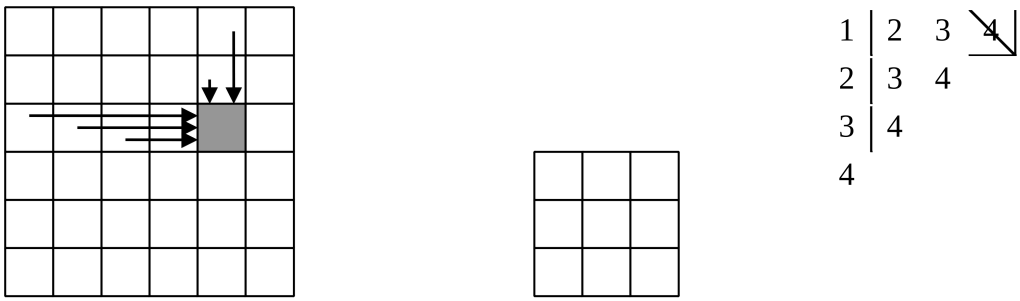
2) $l_{ji} = \dfrac{1}{u_{ii}} (a_{ji} - \sum_{k=1}^{i-1} l_{jk} u_{ki}),\ j = i+1, \ldots, n$

На практике используется компактный способ хранения результата: результирующая матрица содержит обе матрицы:

$$\begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ L_{2,1} & U_{2,2} & U_{2,3} \\ L_{3,1} & L_{3,2} & U_{3,3} \end{pmatrix}$$

При этом матрица L дополняется единицами по главной диагонали. Кроме этого, ввиду особенностей алгоритма, возможно сохранять значения матриц L и U в матрицу A в целях экономии памяти. Именно такой вариант алгоритма и был реализован в качестве теста.

На рис.3.2 показаны зависимости фрагмента данных матрицы U от других фрагментов. Аналогичные зависимости матрицы L симметричны.



а) б) в)

Рис.3.2 Зависимости данных (а, б) и оптимальный план исполнения LU-разложения

Во фрагментированной программе использовались фрагменты нескольких видов: L, U и Diag, вычисляющие, соответственно, значения фрагментов матриц L, U и диагональные фрагменты. Каждый из этих трех фрагментов представлен в двух вариантах – граничном и обычном. Кроме этого использовались фрагменты умножения подматриц MulAdd из предыдущего примера, необходимые для вычисления частичных сумм.

Фрагментированный алгоритм следующий:

$F_{i,i} = D_i : Diag(A_{i,i}, S_{i,i}); i = 1, \ldots, n$

$F_{i,j} = U_{i,j} : L(A_{i,j}, S_{i,j}); i = 1, \ldots, n-1; j = i+1, \ldots, n$

$F_{j,i} = L_{i,j} : L(A_{i,j}, S_{i,j}); i = 2, \ldots, n; j = 1, \ldots, i-1$

${}^{x}G_{i,j} = \{k = 1, \ldots, \min(i, j) - 1 \mid M_{i,j,k}\}; i = 2, \ldots, n; j = 2, \ldots, n$, где

$M_{i,j,k} = MulAdd(A_{i,k}, A_{k,j}, S_{i,j})$, а группа ${}^{x}G_{i,j}$ – эксклюзивная.

${}^{x}G_{i,j} < F_{i,j}; i = 2, \ldots, n; j = 2, \ldots, n$,

${}^{x}G_{i,j} < U_{i-1,j}; i = 2, \ldots, n; j = i+1, \ldots, n$,

${}^{x}G_{i,j} < L_{i,j-1}; i = 2, \ldots, n; j = 2, \ldots, i-1$,

${}^{x}G_{i,j} < D_{\min(i,j)}; i = 2, \ldots, n; j = 2, \ldots, n$,

$D_i < U_{i-1,i}, L_{i,i-1}; i = 2, \ldots, n$

Фрагменты $L_{i,j}$, $U_{i,j}$ и D_i выполняются в соответствии с порядком, указанном на рис.3.2 а) и б). Именно, каждый посчитанный диагональный фрагмент позволяет вычислять любые фрагменты своей вертикали и горизонтали. Однако, такой вариант не всегда будет давать хорошие результаты, так как выполнив все фрагменты вертикали и горизонтали сможет исполняться только один фрагмент – следующий диагональный. Оптимальным порядком исполнения было бы разрастание фронта, как показано на рис.3.2 в).

Исходный текст программы приведен в приложении А. Тестирование проводилось на четырех-процессорной системе архитектуры Intel® Itanium®2 «smp4x64» (см. Приложение Б). Параметрами теста были размер фрагмента данных и количество потоков исполнения. Общий размер данных при этом оставался неизменным: 960×960.

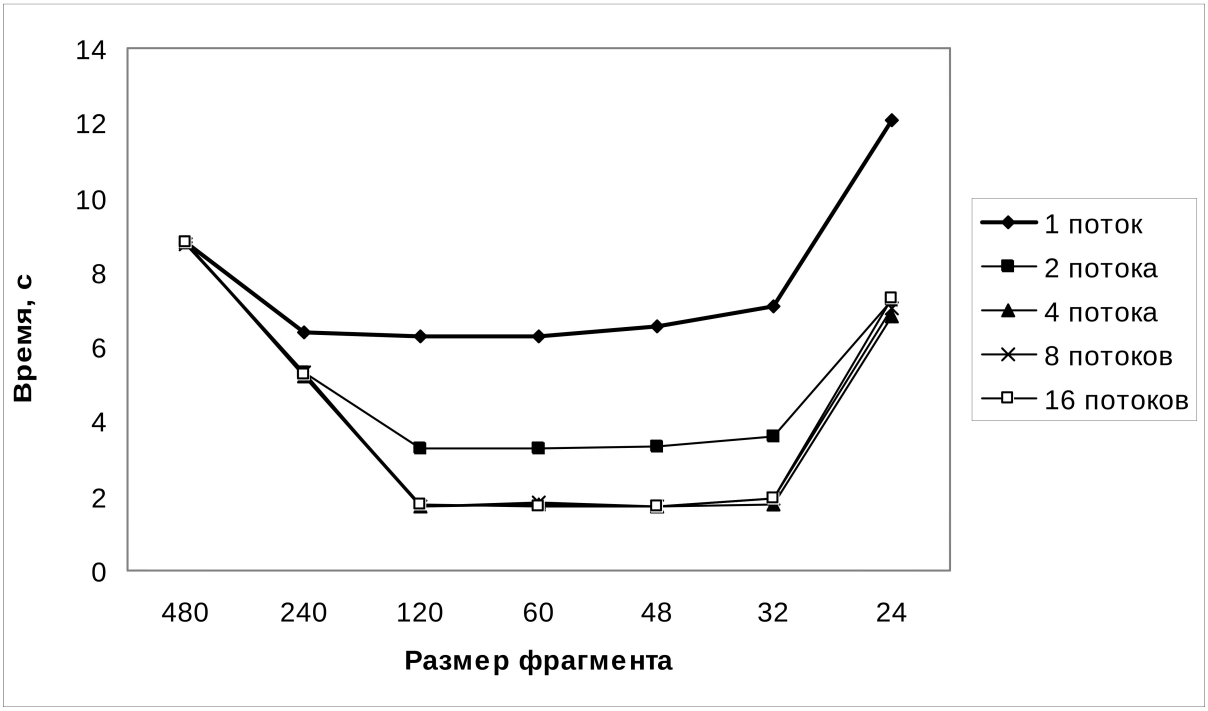


Рис.3.3 Временные характеристики фрагментированного LU-разложения

Картина в целом повторяет тест №1. Отличием является то, что степень параллелизма не достигает 100 процентов, достигая примерно 90 процентов для большинства параметров. Это связано с тем, что множество фрагментов не так хорошо отображается на имеющиеся ресурсы во времени.

Заключение

В диссертации проведен анализ средств параллельного программирования с позиций автоматической реализации прикладных свойств программы. Рассмотрена технология фрагментированного программирования. Реализована run-time система исполнения фрагментированных программ. Проведено ее начальное тестирование, которое продемонстрировало возможность ее практического использования.

На защиту выносятся следующие пункты:

- Модель фрагментированного программирования.
- Способ представления фрагментированных программ.
- Алгоритмы и реализация run-time системы исполнения фрагментированных программ в мультипроцессоре.
- Анализ тестирования run-time системы.

Несмотря на то, что в текущем состоянии run-time систему уже можно использовать для исполнения фрагментированных программ, это только первый шаг в создании системы фрагментированного программирования. Дальнейшее развитие этого проекта планируется по нескольким направлениям:

- Создание языка описания фрагментированных программ.
- Разработка компилятора языка фрагментированного программирования.
- Развитие run-time системы, которое включает усовершенствование внутренних алгоритмов системы, расширение диспетчера до работы в мультикомпьютере, в котором узлом может быть мультипроцессор.
- Создание библиотеки фрагментированных алгоритмов и программ. Сюда относятся библиотека готовых фрагментов и библиотека готовых схем. Это те блоки, из которых конечный пользователь будет составлять свою программу. Такая библиотека практически необходима для реального использования системы фрагментированного программирования.
- Создание визуальной среды разработки, отладки и мониторинга фрагментированных программ. Такая среда позволит создавать и отлаживать фрагментированные программы с помощью интуитивно понятного интерфейса.
- Расширение системы фрагментированного программирования до системы сборочного программирования. Несмотря на самодостаточность, фрагментированное программирование – лишь инструмент, необходимый для реализации более широкой технологии – технологии сборочного

программирования. Именно ее создание и будет логическим продолжением направления.

Литература

1. Вальковский, В.А. Синтез параллельных программ и систем на вычислительных моделях. –Новосибирск: Наука. Сибирское отделение, 1988. – 128 с.
2. Малышкин, В.Э Параллельное программирование мультикомпьютеров. – Изд-во НГТУ, Новосибирск, 2006. – 296 с.
3. Чаюк К. Г. Фрагментированный алгоритм LU-разложения и его параллельная реализация / К. Г. Чаюк ; науч. рук. В. Э. Малышкин // Томский Государственный Университет, сборник докладов участников
4. Чаюк К. Г. Фрагментированный алгоритм LU-разложения и его параллельная реализация / К. Г. Чаюк ; науч. рук. В. Э. Малышкин // Всероссийская научная конференция студентов, аспирантов и молодых учёных. Наука. Технологии. Инновации.
5. Message Passing Interface (MPI) Forum http://www.mpi-forum.org/
6. The OpenMP API specification for parallel programming http://openmp.org/
7. High Performance Fortran Forum (HPFF) home page http://www.crpc.rice.edu/HPFF/home.html
8. Distributed Virtual Machine/Memory http://www.keldysh.ru/dvm
9. Приложения Т-системы http://www.ctc.msiu.ru/program/t-system/
10. Норма. Декларативный язык для спецификации задач вычислительного характера http://parallel.ru/tech/norma/
11. The KeLP Programming System http://www.cs.ucsd.edu/groups/hpcl/scg/kelp/

Приложение А. Листинги фрагментированных программ

Листинг фрагментированной программы умножения матриц

```
#include "rfes.h"

#define FGCOUNT 10
#define FGSIZE 96
#define THREADS 8

using namespace std;

class Zero : public Fragment
{
	int m_Size;
	double *m_Data;

public:
	Zero(Group* Parent, int Priority, double* Data, int Size)
		:Fragment(Parent, Priority)
	{
		m_Size = Size;
		m_Data = Data;
	}

	void Procedure()
	{
		for(int j=0, idx=0; j<m_Size; j++)
			for(int i=0; i<m_Size; i++, idx++)
			{
				m_Data[idx] = 0;
			}
	}
};

class MullAdd : public Fragment
```

```
{
        int m_Size;
        double *m_A, *m_B, *m_C;
public:
        MullAdd(Group* Parent, int Priority, double* A, double* B, double* C, int Size)
                :Fragment(Parent, Priority)
        {
                m_Size = Size;
                m_A = A;
                m_B = B;
                m_C = C;
        }

        void Procedure()
        {
                for(int j=0; j<m_Size; j++)
                        for(int i=0; i<m_Size; i++)
                                for(int k=0; k<m_Size; k++)
                                        m_C[j*m_Size + i] += m_A[j*m_Size + k] *
                                                m_B[k*m_Size + i];
        }
};

int main()
{
        // Create Run-time system
        Rfes r;

        double **A, **B, **C;

        // allocate and initialize matrices
        A = new double*[FGCOUNT*FGCOUNT];
        B = new double*[FGCOUNT*FGCOUNT];
        C = new double*[FGCOUNT*FGCOUNT];
```

```
for(int i=0; i<FGCOUNT; i++)
        for(int j=0; j<FGCOUNT; j++)
        {
                A[i+j*FGCOUNT] = new double[FGSIZE*FGSIZE];
                B[i+j*FGCOUNT] = new double[FGSIZE*FGSIZE];
                for(int x=0; x<FGSIZE; x++)
                        for(int y=0; y<FGSIZE; y++)
                        {
                                A[i+j*FGCOUNT][y*FGSIZE+x] =
                                        1.0/(1.0+x+y);
                                B[i+j*FGCOUNT][y*FGSIZE+x] =
                                        1.0/(1.0+x+y);
                        }
                C[i+j*FGCOUNT] = new double[FGSIZE*FGSIZE];
        }

// Create Fragments and groups and set requirements

for(int i=0; i<FGCOUNT; i++)
        for(int j=0; j<FGCOUNT; j++)
        {
                // Create fragment Zij in SuperGroup
                Zero *z = new Zero(&r.SuperGroup, 0,
                        C[i+j*FGCOUNT], FGSIZE);

                // Create Group Gij in SuperGroup (true for
                // excluding)
                Group *g = new Group(&r.SuperGroup, 0, true);

                // Zij < Gij
                g->AddRequirement(z);
                for(int k=0; k<FGCOUNT; k++)
                        // Add Fijk to Gij
                        MullAdd *F = new MullAdd(g, 0,
```

```
                                        A[k+j*FGCOUNT], B[i+k*FGCOUNT],
                                        C[i+j*FGCOUNT], FGSIZE);
                }
        t.Reset();
        r.Run(THREADS);

}
```

Листинг фрагментированной программы LU-разложения

```
#include <iostream>
#include "rfes.h"
#include "timer.h"

using namespace std;

Timer t;

class L : public Fragment
{
        int Size;
        double *Sum, *Data, *Diag;
        bool Edge;
public:
        L(Group* Parent, int Priority, bool edge, double* sum, double* data, double *diag,
                int size) : Fragment(Parent, Priority)
        {
                Size = size;
                Sum = sum;
                Data = data;
                Diag = diag;
                Edge = edge;
        }
        void Procedure()
        {
                if(Edge)
```

```
                {
                        for(int i=0; i<Size; i++)
                        {
                                Data[i*Size+0] = Data[i*Size+0]/Diag[0*Size+0];
                                for(int j=1; j<Size; j++)
                                {
                                        Data[i*Size+j] -= Sum[i*Size+j];
                                        for(int k=0; k<j; k++)
                                                Data[i*Size+j] -= Data[i*Size+k] * Diag[k*Size+j];
                                        Data[i*Size+j] /= Diag[j*Size+j];
                                }
                        }
                }
                else
                {
                        for(int i=0; i<Size; i++)
                                for(int j=0; j<Size; j++)
                                {
                                        Data[i*Size+j] -= Sum[i*Size+j];
                                        for(int k=0; k<j; k++)
                                                Data[i*Size+j] -= Data[i*Size+k] * Diag[k*Size+j];
                                        Data[i*Size+j] /= Diag[j*Size+j];
                                }
                }
        }
};
class U : public Fragment
{
        int Size;
        double *Sum, *Data, *Diag;
        bool Edge;
public:
        U(Group* Parent, int Priority, bool edge, double* sum, double* data, double *diag,
                int size) : Fragment(Parent, Priority)
        {
```

```
		Size = size;
		Sum = sum;
		Data = data;
		Diag = diag;
		Edge = edge;
	}
	void Procedure()
	{
		if(Edge)
		{
			for(int j=0; j<Size; j++)
				for(int i=1; i<Size; i++)
				{
					Data[i*Size+j] = Data[i*Size+j]-Sum[i*Size+j];
					for(int k=0; k<i; k++)
						Data[i*Size+j] -= Diag[i*Size+k] * Data[k*Size+j];
				}
		}
		else
		{
			for(int j=0; j<Size; j++)
				for(int i=0; i<Size; i++)
				{
					Data[i*Size+j] -= Sum[i*Size+j];
					for(int k=0; k<i; k++)
						Data[i*Size+j] -= Diag[i*Size+k] * Data[k*Size+j];
				}
		}
	}
};

class D : public Fragment
{
	int Size;
	double *Diag, *Sum;
```

```
        bool Edge;
public:
        D(Group* Parent, int Priority, bool edge, double* sum, double* diag, int size)
                : Fragment(Parent, Priority)
        {
                Size = size;
                Diag = diag;
                Sum = sum;
                Edge = edge;

        }
        void Procedure()
        {
                if(Edge)
                {
                        for(int j=1; j<Size; j++)
                        {
                                Diag[j*Size] /= Diag[0*Size+0];
                                for(int i=1; i<=j; i++)
                                {
                                        for(int k=1; k<i; k++)
                                        {
                                                Diag[i*Size+j] -= Diag[i*Size+k]*Diag[k*Size+j];
                                                Diag[j*Size+i] -= Diag[j*Size+k]*Diag[k*Size+i];
                                        }
                                        Diag[j*Size+i] /= Diag[i*Size+i];
                                }
                        }
                }
                else
                {
                        for(int j=0; j<Size; j++)
                        {
                                for(int i=0; i<=j; i++)
                                {
```

```
                                        Diag[i*Size+j] -= Sum[i*Size+j];
                                        for(int k=0; k<i; k++)
                                                Diag[i*Size+j] -= Diag[i*Size+k]*Diag[k*Size+j];

                                        if(i!=j)
                                        {
                                                Diag[j*Size+i] -= Sum[j*Size+i];
                                                for(int k=0; k<i; k++)
                                                        Diag[j*Size+i]                                  -=
Diag[j*Size+k]*Diag[k*Size+i];
                                                Diag[j*Size+i] /= Diag[i*Size+i];
                                        }
                                }
                        }
                }
        }
};

class MullAdd : public Fragment
{
        int m_Size;
        double *m_A, *m_B, *m_C;
public:
        MullAdd(Group* Parent, int Priority, double* A, double* B, double* C, int Size)
                :Fragment(Parent, Priority)
        {
                m_Size = Size;
                m_A = A;
                m_B = B;
                m_C = C;
                m_EraseAfterExec = true;
        }

        void Procedure()
        {
```

```
                for(int j=0; j<m_Size; j++)
                        for(int i=0; i<m_Size; i++)
                                for(int k=0; k<m_Size; k++)
                                        m_C[j*m_Size + i] += m_A[j*m_Size + k] *
                                                m_B[k*m_Size + i];
        }
};

#define FGCOUNT 40
#define FGSIZE (960/FGCOUNT)
#define SZ (FGCOUNT*FGSIZE)

int THREADS;

int main(int argc, char** argv)
{
        Rfes r;
        double **A, **Sum;

        if(argc!=2 || sscanf(argv[1], "%d", &THREADS)!=1)
                THREADS=1;

        A = new double*[FGCOUNT*FGCOUNT];
        Sum = new double*[FGCOUNT*FGCOUNT];

        for(int i=0; i<FGCOUNT; i++)
                for(int j=0; j<FGCOUNT; j++)
                {
                        A[i*FGCOUNT+j] = new double[FGSIZE*FGSIZE];
                        Sum[i*FGCOUNT+j] = new double[FGSIZE*FGSIZE];
                        for(int y=0; y<FGSIZE; y++)
                                for(int x=0; x<FGSIZE; x++)
                                {
                                        A[i*FGCOUNT+j][y*FGSIZE+x] = 1.0/
                                                (1.0+x+y+i*FGSIZE+j*FGSIZE);
```

```
                                        Sum[i*FGCOUNT+j][y*FGSIZE+x] = 0;
                                }
                        }
                Fragment** f = new Fragment*[FGCOUNT*FGCOUNT];
                Group** g = new Group*[FGCOUNT*FGCOUNT];

                for(int i=0; i<FGCOUNT; i++)
                        for(int j=0; j<FGCOUNT; j++)
                        {
                                if(i*j>0)
                                {
                                        g[i*FGCOUNT+j] = new Group(&r.SuperGroup, +i+j, true);
                                        for(int k=0; k<i && k<j; k++)
                                        {
                                                MullAdd *m = new MullAdd(g[i*FGCOUNT+j], +i+j,
A[i*FGCOUNT+k],
                                                        A[k*FGCOUNT+j], Sum[i*FGCOUNT+j],
FGSIZE);
                                                m->AddRequirement(f[i*FGCOUNT+k]);
                                                m->AddRequirement(f[k*FGCOUNT+j]);
                                        }
                                }

                                if(i==j)
                                        f[i*FGCOUNT+j] = new D(&r.SuperGroup, +i+j, i==0,
Sum[i*FGCOUNT+j],
                                                A[i*FGCOUNT+j], FGSIZE);
                                else if (i<j)
                                {
                                        f[i*FGCOUNT+j] = new U(&r.SuperGroup, +i+j, i==0,
Sum[i*FGCOUNT+j],
                                                A[i*FGCOUNT+j], A[i*FGCOUNT+i], FGSIZE);
                                        f[i*FGCOUNT+j]->AddRequirement(f[i*FGCOUNT+i]);
                                }
                                else
```

```
                        {
                                f[i*FGCOUNT+j]  =  new  L(&r.SuperGroup,  +i+j,  j==0,
Sum[i*FGCOUNT+j],
                                        A[i*FGCOUNT+j], A[j*FGCOUNT+j], FGSIZE);
                                f[i*FGCOUNT+j]->AddRequirement(f[j*FGCOUNT+j]);
                        }
                        if(i*j>0)
                                f[i*FGCOUNT+j]->AddRequirement(g[i*FGCOUNT+j]);
                }

        t.Reset();
        r.Run(THREADS);
        cout << "FgCount=" << FGCOUNT << " FgSize=" << FGSIZE <<
                " Threads=" << THREADS << " Time=" << t.GetMillis() << endl;
        return 0;
}
```

Приложение Б. Описание используемого аппаратного обеспечения

Название	Характеристика
smp4x64	Вычислительный узел HP Integrity rx4640, 4 процессора Itanium2, 1.5 GHz, Кэш-память: L1 16 KB, L2 256 KB, L3 4 MB. Оперативная память: 64 GB.