

**Л. К. Птицына Е. В. Дорофеева**

# **ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ КОМПЬЮТЕРНЫХ СЕТЕЙ**

**Разработка программного обеспечения в базисе  
функций библиотеки MPI для оценки динамических  
характеристик параллельных программ**

**Учебное пособие**

**Санкт-Петербург  
Издательство Политехнического университета  
2006**



*Л.К. Птицына Е.В. Дорофеева*

# **ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ КОМПЬЮТЕРНЫХ СЕТЕЙ**

**Разработка программного обеспечения в базисе функций  
библиотеки MPI для оценки динамических характеристик  
параллельных программ**

**Учебное пособие**

Санкт-Петербург  
Издательство Политехнического университета  
2006

УДК 004.77:004.451.62.032.24 (075.8)

ББК 32.973.202 – 018.2 я73

П874

Птицына Л.К., Дорофеева Е.В. **Программное обеспечение компьютерных сетей. Разработка программного обеспечения в базисе функций библиотеки MPI для оценки динамических характеристик параллельных программ:** Учеб. пособие. СПб.: Изд-во Политехн. ун-та, 2006. 248 с.

Учебное пособие соответствует содержанию дисциплины национально-регионального (вузовского) компонента «Программное обеспечение компьютерных сетей» государственного образовательного стандарта по направлению подготовки бакалавров и магистров 220100 «Системный анализ и управление» и по специальности высшего профессионального образования 230201 «Информационные системы и технологии». В учебном пособии описаны общие тенденции развития современных подходов к разработке и исследованию программного обеспечения для высокопроизводительных вычислительных систем. Определена позиция библиотеки MPI в составе средств программирования параллельных вычислений. Сформированы основные функциональные группы функций библиотеки MPI и определены их ключевые особенности. Раскрыты приемы создания научно-обоснованной системы прототипов ключевых программных компонентов в базисе функций библиотеки MPI для оценки качества работы программного обеспечения высокопроизводительных вычислительных систем. Предназначено для студентов старших курсов обучения по дисциплине «Программное обеспечение компьютерных сетей» направления 220100 «Системный анализ и управление» и специальности «Информационные системы и технологии». Может быть использовано при подготовке студентов высших учебных заведений, обучающихся по направлениям «Информационные технологии», «Прикладная математика и информатика», «Автоматизация и управление», «Информатика и вычислительная техника», «Информационные системы» и по специальностям «Вычислительные машины, комплексы, системы и сети», «Программное обеспечение вычислительной техники и автоматизированных систем».

Табл. 18. Ил. 9. Библиогр.: 47 назв.

ISBN 5-7422-1352-2

© Птицына Л.К., Дobreцов С.В., 2006

© Санкт-Петербургский государственный  
политехнический университет, 2006

## Содержание

ВВЕДЕНИЕ.....	6
1. СРАВНИТЕЛЬНЫЙ АНАЛИЗ СОВРЕМЕННЫХ ПОДХОДОВ К РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ.....	11
1.1. Современные подходы к разработке параллельных программ .....	11
1.2. Модели и языки параллельного программирования.....	14
1.2.1. Модель передачи сообщений .....	15
1.2.2. Модель параллелизма по данным.....	17
1.2.3. Гибридная модель параллелизма по управлению с передачей сообщений.....	18
1.2.4. Модель параллелизма по данным и управлению.....	21
1.3. Сопоставление возможностей стандартных подходов к разработке и сопровождению параллельных программ .....	24
1.3.1. Адекватность модели и языка программирования .....	25
1.3.2. Методы и средства отладки .....	26
1.3.3. Сопровождаемые варианты программы .....	28
1.4. Эффективность выполнения параллельных программ по тестам NBP.....	28
1.5. Переносимость и повторное использование параллельных программ.....	29
1.6. Выводы.....	32
2. ХАРАКТЕРИСТИКА СПЕЦИФИКАЦИИ БАЗИСА ФУНКЦИЙ БИБЛИОТЕКИ MPI.....	35
2.1. Общая организация библиотеки MPI.....	35
2.2. Базовые функции библиотеки MPI.....	37
2.3. Коммуникационные операции типа точка-точка.....	38
2.3.1. Обзор коммуникационных операций типа точка-точка .....	38
2.3.2. Блокирующие коммуникационные операции .....	40
2.3.3. Неблокирующие коммуникационные операции .....	41
2.4. Коллективные операции.....	42
2.4.1. Обзор коллективных операций .....	42
2.4.2. Функции сбора блоков данных от всех процессов группы.....	44
2.4.3. Функции распределения блоков данных по всем процессам группы .....	45
2.4.4. Совмещенные коллективные операции .....	45
2.4.5. Глобальные вычислительные операции над распределенными данными.....	45
2.5. Производные типы данных и передача упакованных данных.....	46
2.5.1. Производные типы данных .....	46
2.5.2. Передача упакованных данных.....	48
2.6. Сравнение версий библиотеки MPI-1.1 и MPI-2.....	48
2.7. Выводы.....	49

<b>3. РАЗРАБОТКА ПРОТОТИПОВ БАЗОВЫХ КОМПОНЕНТОВ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ В БАЗИСЕ ФУНКЦИЙ МРІ ДЛЯ ОЦЕНКИ СТАТИСТИЧЕСКИХ ХАРАКТЕРИСТИК ВРЕМЕНИ ВЫПОЛНЕНИЯ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ.....</b>	<b>51</b>
3.1. Типовые процедуры оценки статистических характеристик времени выполнения параллельных программ.....	52
3.2. Обоснование выбора технических средств.....	57
3.3. Программный прототип объединения подпрограмм по логической функции «И».....	59
3.3.1. Функциональная спецификация прототипа.....	59
3.3.2. Альтернативные варианты реализации программного прототипа.....	62
3.3.3. Сравнение приведенных топологий .....	65
3.3.4. Построение схемы измерения временных характеристик .....	66
3.3.5. Методика тестирования, проведение эксперимента и представление результатов.....	67
3.4. Программный прототип объединения подпрограмм по логической функции «ИЛИ».....	78
3.4.1. Функциональная спецификация прототипов.....	78
3.4.2. Альтернативные варианты реализации программного прототипа.....	80
3.4.3. Методика тестирования и проведение эксперимента.....	82
3.5. Программный прототип объединения подпрограмм по логическим функциям «И–ИЛИ».....	84
3.5.1. Функциональная спецификация прототипа.....	85
3.5.2. Построение схемы измерения временных характеристик .....	86
3.5.3. Методика тестирования, проведение эксперимента и представление результатов.....	87
3.6. Выводы.....	90
<b>4. АНАЛИЗ РАБОТОСПОСОБНОСТИ ПРОТОТИПОВ БАЗОВЫХ КОМПОНЕНТОВ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.....</b>	<b>91</b>
4.1. Подтверждение работоспособности программного прототипа объединения подпрограмм по логической функции «И».....	91
4.1.1. Последовательная программа .....	91
4.1.2. Построение вычислений в пакете MathCad .....	96
4.2. Подтверждение работоспособности программного прототипа объединения подпрограмм по логической функции «ИЛИ».....	97
4.2.1. Последовательная программа .....	97
4.2.2. Построение вычислений в пакете MathCad .....	101
4.3. Выводы.....	102
<b>5. ОЦЕНКА СТАТИСТИЧЕСКИХ ХАРАКТЕРИСТИК ОБНАРУЖЕНИЯ И ОТРАЖЕНИЯ УГРОЗ ДЛЯ АВТОМАТИЗИРОВАННЫХ СИСТЕМ ТРЕТЬЕГО КЛАССА ЗАЩИЩЕННОСТИ В БАЗИСЕ ФУНКЦИЙ МРІ.....</b>	<b>103</b>
5.1. Базовая модель процессов обнаружения и отражения угроз для автоматизированных систем третьего класса защищенности.....	103

5.2. Вывод аналитических соотношений для определения динамических характеристик.....	111
5.3. Описание программы.....	123
5.3.1. Работа с программой.....	123
5.3.2. Результаты экспериментов.....	125
5.4. Выводы.....	126
ЗАКЛЮЧЕНИЕ.....	128
БИБЛИОГРАФИЧЕСКИЙ СПИСОК.....	131

Приложение 1.

Описание стандарта MPI 1.1.....	134
---------------------------------	-----

Приложение 2.

Исходные тексты программ прототипов.....	174
--	-----

Приложение 3.

Программная документация на программное обеспечение для определения статистических характеристик обнаружения и отражения угроз автоматизированных систем третьего класса защищенности.....	193
--	-----

## **ВВЕДЕНИЕ**

Непрерывно расширяющиеся границы областей применения высокопроизводительных вычислительных систем и жесткие условия конкурентной борьбы за потребителя обуславливают устойчивое развитие их программного обеспечения. Одно из перспективных направлений развития программного обеспечения связано с реализацией интеллектуальных функций, позволяющих отслеживать и регулировать качество его работы. Однако подобное существующее программное обеспечение не содержит компонентов, позволяющих оценить их динамические характеристики в реальных условиях.

В существующем многообразии разновидностей моделей параллельного программирования основной является модель передачи сообщений, поскольку ей отдается предпочтение при основном приоритете снижения временных затрат на выполнение параллельных вычислений. Коммуникационная библиотека MPI стала общепризнанным стандартом в параллельном программировании с использованием механизма передачи сообщений. При выбранной модели параллельного программирования процесс определения качества функционирования программ отчужден от ее выполнения. Качество функционирования разработанных программ выявляется при тестировании. Одновременно с этим, последние достижения в области аналитического определения динамических характеристик параллельных программ и распределенных приложений предоставляют математическую основу для контроля качества их функционирования.

В связи с этим целью пособия является раскрытие приемов создания научно-обоснованной системы прототипов ключевых программных компонентов в базисе функций библиотеки MPI для оценки качества работы программного обеспечения высокопроизводительных вычислительных систем.

Согласно указанной цели в пособии решаются следующие задачи:

1. Определение характеристики спецификаций базиса функций библиотеки MPI и детальное рассмотрение ключевых особенностей их применения.
2. Разработка прототипов ключевых компонентов программного обеспечения в базисе функций библиотеки MPI для оценки статистических характеристик времени выполнения параллельных программ.

3. Выполнение анализа работоспособности ключевых компонентов программного обеспечения.
4. Разработка прототипа программы для оценки статистических характеристик обнаружения и отражения угроз для автоматизированных систем третьего класса защищенности в базисе функций MPI.

В данном пособии представляется новый путь развития современных подходов к разработке программного обеспечения для высокопроизводительных вычислительных систем в классе моделей передачи сообщений параллельно выполняемых программ на основе интеллектуализации процессов анализа их качества.

Предусматриваемая интеллектуализация ориентирована на включение в функциональную спецификацию параллельных программ компонент, предназначенных для оценки динамических характеристик.

В первой главе выполнен сравнительный анализ современных подходов к разработке программного обеспечения для высокопроизводительных вычислительных систем. При сравнительном анализе раскрыты особенности модели передачи сообщений, модели параллелизма по данным, гибридной модели параллелизма по данным и управлению; выделены преимущества и недостатки современных моделей и языков программирования. При сопоставлении возможностей стандартных подходов к разработке и сопровождению параллельных программ рассмотрены вопросы адекватности модели и языка программирования, методы и средства отладки. По тестам NBP проанализирована эффективность выполнения параллельных программ. Выделены ключевые аспекты переносимости и повторного использования параллельных программ. На основе результатов сравнительного анализа современных подходов к разработке программного обеспечения для высокопроизводительных вычислительных систем определены цель и задачи работы.

Во второй главе охарактеризована спецификация базиса функций библиотеки MPI. Определена общая организация библиотеки MPI, выделены ее базовые функции, описаны коммуникационные операции типа точка – точка и коллективные операции, рассмотрены характерные особенности производных типов данных и передачи упакованных данных. В конце второй главы проведено сопоставление версий библиотеки MPI-1.1 и MPI-2.

Третья глава посвящена разработке прототипов ключевых компонентов программного обеспечения в базисе функций библиотеки MPI для оценки статистических характеристик времени выполнения параллельных программ. Первоначально выделены типовые процедуры оценки статистических характеристик времени выполнения параллельных программ и обоснован выбор технических средств. Затем представлены основные этапы разработки программных прототипов. При представлении описаны функциональная спецификация, альтернативные варианты реализации программного прототипа, результаты проведенного сравнения альтернативных топологий, схема измерения временных характеристик, разработанная методика тестирования.

В четвертой главе раскрыто содержание выполненного анализа работоспособности прототипов базовых компонентов программного обеспечения. При подтверждении корректности функционирования базовых компонентов задействован пакет MathCad.

В пятой главе рассмотрен полный цикл оценки статистических характеристик обнаружения и отражения угроз для автоматизированных систем третьего класса защищенности в базисе функций MPI. Сначала построена базовая модель процессов обнаружения и отражения угроз для автоматизированных систем третьего класса защищенности. Затем выполнен вывод аналитических соотношений для определения динамических характеристик. Созданное математическое обеспечение послужило основой для разработанного программного прототипа, нацеленного на оценку качества работы систем защиты. В конце пятой главы приведено описание программы, включающее общие сведения, функциональное назначение, логическую структуру, используемые технические средства, входные данные, выходные данные, вызов и загрузку, план проведения эксперимента, результаты экспериментов.

В заключении выделены основные результаты решения представленных задач, их научная и практическая значимость.

В Приложении 1 приведено расширенное описание основных функций MPI. В Приложении 2 представлены тексты программ для оценки элементарных динамических характеристик параллельных программ. В Приложении 3 приводится программная документация на программу для определения статистических характеристик обнаружения и отражения угроз автоматизированных систем третьего класса защищенности.

Описание одной из представленных программ выполнено в соответствии с требованиями к оформлению магистерских диссертаций.

Научную значимость материалов пособия имеют следующие результаты:

1. Новый подход к формированию шаблонов для программного обеспечения высокопроизводительных вычислительных систем, способного регулировать качество выполняемых задач в среде MPI.
2. Расширенный состав моделей, описывающих функционирование систем третьего класса защищенности с позиций информационной безопасности.
3. Развитое математическое обеспечение для определения динамических характеристик систем защиты информации, соответствующих третьему классу защищенности в условиях последовательно-параллельной и распределенной обработки информации.

Практическое применение основных результатов работы могут найти разработанные программные прототипы для создания интеллектуального программного обеспечения, способного динамически регулировать качество собственного функционирования.

Приведенное детальное описание функций библиотеки MPI расширяет состав информационного обеспечения образовательных программ подготовки бакалавров и магистров по направлениям «Информационные технологии», «Прикладная математика и информатика», «Системный анализ и управление», «Автоматизация и управление», «Информатика и вычислительная техника», «Информационные системы».

Предложенные способы организации прототипов учитывают ключевые особенности типовых топологий параллельных вычислений, применяемых на практике. Проведенный сравнительный анализ обосновывает отбор наилучшего варианта с точки зрения наименьших затрат времени на их выполнение.

Построенная система зависимостей, демонстрирующая влияние характеристик параллельных программ на динамически формируемые оценки, позволяет обосновать рациональный выбор размерности входных данных.

Представленные в пособии материалы используются при проведении лабораторных работ, практических занятий и научно-исследовательских работ со студентами факультета технической кибернетики и физико-механического факультета с 2002 года.

Приведенные в пособии результаты, обладающие научной новизной, представлены на X Всероссийской конференции по проблемам науки и высшей школы «Фундаментальные исследования в технических университетах» в Санкт-Петербургском государственном политехническом университете (18-19 мая 2006 г., Санкт-Петербург); X Международной научно-практической конференции «Системный анализ в проектировании и управлении» в Санкт-Петербургском государственном политехническом университете, Таганрогском государственном радиотехническом университете (28 июня – 10 июля 2006 г., Санкт-Петербург, Таганрог); Межвузовской научно-методической конференции «Инновационные и научноемкие технологии в высшем образовании России» в Московском государственном институте радиотехники, электроники и автоматики (техническом университете) (30 мая 2006 г., Москва) [43 – 47].

# **1. СРАВНИТЕЛЬНЫЙ АНАЛИЗ СОВРЕМЕННЫХ ПОДХОДОВ К РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ**

В данном разделе выделены ключевые особенности современных подходов к разработке программного обеспечения для высокопроизводительных вычислительных систем и предложены новые пути в классе моделей передачи сообщений параллельно выполняемых программ на основе интеллектуализации процессов мониторинга их динамических характеристик.

При проведении сравнительного анализа современных подходов к разработке программного обеспечения для высокопроизводительных вычислительных систем систематизированы и расширены представления об их сущности, преимуществах и недостатках.

В ходе систематизации использованы опубликованные и апробированные материалы ведущих специалистов по программному обеспечению высокопроизводительных вычислительных систем [1 – 31]. В заключении данного раздела обоснована актуальность проводимых исследований.

## **1.1. Современные подходы к разработке параллельных программ**

Прошло немногим более 50 лет с момента появления первых электронных вычислительных машин – компьютеров. За это время сфера их применения охватила практически все области человеческой деятельности.

Сегодня невозможно представить эффективную организацию работы без применения компьютеров в таких областях, как планирование и управление производством, проектирование и разработка сложных технических устройств, издательская деятельность, образование – словом, во всех областях, где возникает необходимость в обработке больших объемов информации. Однако наиболее важным по-прежнему остается использование их в том направлении, для которого они собственно и создавались, а именно, для решения задач, требующих выполнения громадных объемов вычислений. Такие задачи возникли в середине прошлого века в связи с развитием атомной энергетики,

авиастроения, ракетно-космических технологий и ряда других областей науки и техники.

Круг задач, требующих для своего решения применения мощных вычислительных ресурсов, продолжает расширяться. Это связано с тем, что произошли фундаментальные изменения в самой организации научных исследований. Вследствие широкого внедрения вычислительной техники значительно усилилось направление численного моделирования и численного эксперимента. Численное моделирование, заполняя промежуток между физическими экспериментами и аналитическими подходами, позволило изучать явления, которые являются либо слишком сложными для исследования аналитическими методами, либо слишком дорогостоящими или опасными для экспериментального изучения. При этом численный эксперимент позволил значительно удешевить процесс научного и технологического поиска. Стало возможным моделировать в реальном времени процессы интенсивных физико-химических и ядерных реакций, глобальные атмосферные процессы, процессы экономического и промышленного развития регионов. Очевидно, что решение таких масштабных задач требует значительных вычислительных ресурсов [1].

В последние годы во всем мире происходит бурное внедрение вычислительных систем с архитектурой, отличной от традиционной последовательной неймановской архитектуры, и новых принципов обработки информации и вычислений, а значит развиваются инструментальные средства, обеспечивающие описание параллельных алгоритмов и их выполнение на вычислительных системах нетрадиционных параллельных архитектур. Появляющиеся системы представляют собой, как правило, отдельные узлы (компьютеры) связанные между собой коммуникационной системой, построенной в соответствии с определенной телекоммуникационной технологией.

Для повышения уровня соотношения производительность/стоимость используются процессоры, которые отличаются не только по производительности, но и по архитектуре. Поэтому большинство систем носит неоднородный характер, что порождает серьезные проблемы:

- Различие производительности процессоров требует распределения работы между процессами, выполняющимися на разных процессорах;
- Различие в архитектуре процессоров требует подготовки разных выполняемых файлов для разных узлов, а различия в представлении

данных потребует и преобразования информации при передаче сообщений между узлами;

- Изменения нагрузки сети добавляют к возможной коммуникационной среды неоднородность, которую тоже необходимо учитывать.

Все это требует гораздо более высокого уровня автоматизации разработки параллельных программ, чем тот, который доступен в настоящее время прикладным программистам.

С 1992 года, когда мультикомпьютеры стали самыми производительными вычислительными системами, резко возрос интерес к проблеме разработки для них параллельных прикладных программ. К этому моменту уже стало ясно, что трудоемкость разработки прикладных программ для многопроцессорных систем с распределенной памятью является главным препятствием для их широкого внедрения. За прошедший с тех пор период предложено много различных подходов к разработке параллельных программ, созданы десятки различных языков параллельного программирования и множество различных инструментальных средств. Среди них специалисты выделяют следующие интересные разработки – Норма [2], Fortran–GNS [3], Fortran–DVM [4], mpC [5], Т–система [6].

Наиболее ярко различные подходы к разработке параллельных программ для вычислительных кластеров и сетей отражают следующие средства: MPI [7, 8], HPF [9, 10], OpenMP + MPI [11] и DVM [4, 12].

Выбор для анализа именно этих подходов объясняется следующими соображениями:

- Все эти подходы ориентированы на программистов, использующих стандартные языки Фортран или С. Именно эти программисты и разрабатывают, в основном, параллельные вычислительные программы.
- В основе этих подходов лежат существенно различающиеся модели и языки параллельного программирования.
- На базе этих подходов созданы инструментальные средства разработки параллельных программ, доступные на многих аппаратных платформах.
- Для всех этих подходов имеется информация об эффективности выполнения соответствующих реализаций тестов, что позволяет объективно судить о пригодности указанных подходов для разработки сложных параллельных программ.

Предпочтительность использования того или иного подхода определяют следующие факторы:

- Легкость разработки и сопровождения параллельных программ;
- Эффективность выполнения параллельных программ;
- Переносимость и повторное использование параллельных программ.

## 1.2. Модели и языки параллельного программирования

Разработкой, анализом и использованием моделей и языков параллельного программирования занимается достаточно представительное сообщество специалистов [1 – 31].

Языковые средства для разработки программного обеспечения высокопроизводительных вычислительных систем соответствуют определенным моделям параллельного программирования.

В настоящее время выделяют следующие разновидности моделей параллельного программирования:

- модель передачи сообщений;
- модель параллелизма по данным;
- гибридная модель параллелизма по управлению с передачей сообщений;
- модель параллелизма по данным и управлению.

Основной моделью параллельного программирования для вычислительных систем является модель передачи сообщений. В этой модели параллельная программа представляет собой систему процессов, взаимодействующих посредством передачи сообщений.

Для построения языка программирования, в основу которого положена модель передачи сообщений, используют следующие варианты:

- Расширение стандартного языка последовательного программирования библиотечными функциями (например, Фортран + MPI, С + MPI);
- Расширение стандартного языка последовательного программирования специальными конструкциями (например, Fortran – GNS);
- Разработка нового языка (например, Occam).

Модель передачи сообщений является слишком низкоуровневой, непривычной и неудобной для программистов, разрабатывающих вычислительные программы. При этой модели программист имеет дело с параллельными процессами и низкоуровневыми примитивами передачи сообщений.

Поэтому вполне естественно, что прикладной программист отдал бы предпочтение тому инструменту, который автоматически преобразует последовательную программу в параллельную программу для вычислительной системы. К сожалению, такое автоматическое распараллеливание в настоящее время невозможно в силу следующих причин:

- 1) Взаимодействие процессоров через коммуникационную систему требует значительного времени, следовательно, вычислительная работа должна распределяться между процессорами крупными порциями. Укрупнение распределяемых порций работы требует анализа более крупных фрагментов программы, обычно включающих в себя вызовы различных процедур. Это, в свою очередь, требует сложного межпроцедурного анализа. С увеличением порций распределяемой работы увеличивается вероятность того, что конструкции, которые на самом деле допускают параллельное выполнение, могут быть пропущены.
- 2) На системах с распределенной памятью необходимо произвести не только распределение вычислений, но и распределение данных, а также обеспечить на каждом процессоре доступ к удаленным данным.
- 3) Распределение вычислений и данных должно быть произведено согласованно. Несогласованность распределения вычислений и данных приводит, к тому, что параллельная программа выполняется значительно медленнее последовательной.

### 1.2.1. Модель передачи сообщений

В модели передачи сообщений параллельная программа представляет собой множество процессов, каждый из которых имеет собственное локальное адресное пространство. Взаимодействие процессов—обмен данными и синхронизация—осуществляется посредством передачи сообщений.

Обобщение и стандартизация различных библиотек передачи сообщений привели в 1993 году к разработке стандарта MPI (Message Passing Interface). Его широкое внедрение в последующие годы обеспечило проявление коренного перелома в решении проблемы переносимости параллельных программ, разрабатываемых в рамках разных подходов, использующих модель передачи сообщений в качестве модели выполнения.

В числе основных достоинств MPI по сравнению с интерфейсами других коммуникационных библиотек обычно указывают следующие его возможности:

- Возможность использования в языках Фортран, С, С++;
- Предоставление возможностей для совмещения обменов сообщениями и вычислений;
- Предоставление режимов передачи сообщений, позволяющих избежать излишнего копирования информации для буферизации;
- Широкий набор коллективных операций (например, широковещательная рассылка информации, сбор информации с разных процессоров), допускающих более эффективную реализацию, чем использование соответствующей последовательности пересылок точка-точка;
- Широкий набор редукционных операций (например, суммирование расположенных на разных процессорах данных или нахождение их максимальных или минимальных значений), не только упрощающих работу программиста, но и допускающих гораздо более эффективную реализацию, чем это может сделать прикладной программист, не имеющий информации о характеристиках коммуникационной системы;
- Удобные средства именования адресатов сообщений, упрощающие разработку стандартных программ или разделение программы на функциональные блоки;
- Возможность задания типа передаваемой информации, что позволяет обеспечить ее автоматическое преобразование в случае различий в представлении данных на разных узлах системы.

Однако разработчики MPI подвергаются и суровой критике за то, что интерфейс получился слишком громоздким и сложным для прикладного программиста [15]. Интерфейс оказался сложным и для реализации. В итоге, в настоящее время практически не существует реализаций MPI, в которых в полной мере обеспечивается совмещение обменов с вычислениями.

Появившийся в 1997 проект стандарта MPI-2 [8] выглядит еще более громоздким и неподъемным для полной реализации. Он предусматривает развитие в следующих направлениях:

- Динамическое создание и уничтожение процессов;

- Односторонние коммуникации и средства синхронизации для организации взаимодействия процессов через общую память (для эффективной работы на системах с непосредственным доступом процессоров к памяти других процессоров);
- Параллельные операции ввода-вывода (для эффективного использования существующих возможностей параллельного доступа многих процессоров к различным дисковым устройствам).

### 1.2.2. Модель параллелизма по данным

В модели параллелизма по данным отсутствует понятие процесса и, как следствие, явная передача сообщений или явная синхронизация. В этой модели данные последовательной программы распределяются программистом по процессорам параллельной машины. Последовательная программа преобразуется компилятором в параллельную программу, выполняющуюся в модели передачи сообщений. При этом вычисления распределяются по правилу собственных вычислений: каждый процессор выполняет только вычисления собственных данных, т.е. данных, распределенных на этот процессор.

Модель параллелизма по данным имеет следующие достоинства:

- Параллелизм по данным является естественным параллелизмом вычислительных задач, поскольку для них характерно вычисление по одним и тем же формулам множества однотипных величин – элементов массивов.
- В модели параллелизма по данным сохраняется последовательный стиль программирования. Программист не должен представлять программу в виде взаимодействующих процессов и заниматься низкоуровневым программированием передач сообщений и синхронизации.
- Распределение вычисляемых данных между процессорами – это не только самый компактный способ задать распределение работы между процессорами, но и способ повышения локализации данных. Чем меньше данных требуется процессору для выполнения возложенной на него работы, тем быстрее она будет выполнена.

Обобщение и стандартизация моделей параллелизма по данным привели к созданию в 1993 году стандарта HPF (High Performance Fortran) – расширения языка Фортран 90. Аналогичные расширения предложены и для языка С и С++.

Проведем анализ основных возможностей HPF.

Прежде всего, программист должен распределить данные между процессорами. Это распределение производится в два этапа. Сначала с помощью директивы ALIGN задается соответствие между взаимным расположением элементов нескольких массивов, а затем вся эта группа массивов с помощью директивы DISTRIBUTE отображается на решетку процессоров. Это отображение, например, может осуществляться следующим образом: каждый массив разрезается несколькими гиперплоскостями на секции примерно одинакового объема, каждая из которых будет расположена на своем процессоре. Заданное распределение данных может быть изменено на этапе выполнения программы с помощью операторов REALIGN и REDISTRIBUTE.

Многие встроенные функции, которые имеют дело с массивами (например, редукционные функции), могут выполняться параллельно.

Безусловно, по сравнению с MPI язык HPF намного упрощает написание параллельных программ, но и его реализация требует от компилятора очень высокого интеллекта. Конечно, самая сложная часть работы, которая вызывала проблемы при автоматическом распараллеливании – распределение данных – возлагается теперь на программиста. Но, и с оставшейся частью работы компилятор не всегда способен справиться без дополнительных решений программиста. Некоторые такие решения были включены в HPF, но все равно оставались серьезные сомнения относительно эффективности HPF-программ. К сожалению, эти сомнения оказались не напрасными. В течение нескольких лет так и не удалось создать компилятора с приемлемой эффективностью.

В 1997 году появился проект стандарта HPF2 [10], в котором существенно расширены возможности программиста по спецификации тех свойств его программы, извлечь которые на этапе компиляции очень трудно или даже вообще невозможно.

### 1.2.3. Гибридная модель параллелизма по управлению с передачей сообщений

Модель параллелизма по управлению (work-sharing model) возникла как модель программирования для мультипроцессоров [16]. На мультипроцессорах в качестве модели выполнения используется модель общей памяти. В этой модели параллельная программа представляет собой систему нитей, взаимодействующих посредством общих переменных и примитивов синхронизации. Нить ("thread") – это

легковесный процесс, имеющий с другими нитями общие ресурсы, включая общую оперативную память.

Основная идея модели параллелизма по управлению заключается в следующем. Вместо программирования в терминах нитей предлагалось расширить языки специальными управляющими конструкциями – параллельными циклами и параллельными секциями. Создание и уничтожение нитей, распределение между ними витков параллельных циклов или параллельных секций (например, вызовов процедур) – все это возлагается на компилятор.

Первая попытка стандартизовать такую модель привела к появлению в 1990 году проекта языка PCF Fortran (проект стандарта X3H5). Однако этот проект тогда не привлек широкого внимания и, фактически, остался только на бумаге. Возможно, что причиной этого было снижение интереса к мультипроцессорам и всеобщее увлечение мультикомпьютерами и HPF.

Однако спустя несколько лет ситуация сильно изменилась. Во-первых, успехи в развитии элементной базы сделали очень перспективным и экономически выгодным создавать мультипроцессоры. Во-вторых, широкое развитие получили мультикомпьютеры с DSM (distributed shared memory – распределенная общая память), позволяющие программам на разных узлах взаимодействовать через общие переменные также, как и на мультипроцессорах. В-третьих, не оправдались надежды на то, что HPF станет фактическим стандартом для разработки вычислительных программ.

Крупнейшие производители компьютеров и программного обеспечения объединили свои усилия и в октябре 1997 года выпустили описание языка OpenMP Fortran – расширение языка Фортран 77. Позже вышли аналогичные расширения языков С и Фортран 90/95.

Проведем анализ основных возможностей OpenMP. OpenMP – это интерфейс прикладной программы, расширяющий последовательный язык программирования набором директив компилятора, вызовом функций библиотеки поддержки выполнения и переменных среды.

Программа начинает свое выполнение как один процесс, называемый главной нитью. Главная нить выполняется последовательно, пока не встретится первая параллельная область программы. Параллельная область определяется парой директив PARALLEL и END PARALLEL. При входе в параллельную область главная нить порождает некоторое число подчиненных ей нитей, которые вместе с ней образуют текущую группу нитей. Все операторы программы, находящиеся в

параллельной конструкции, включая и вызываемые изнутри нее процедуры, выполняются всеми нитями текущей группы параллельно, пока не произойдет выход из параллельной области или встретится одна из конструкций распределения работы – DO, SECTIONS или SINGLE.

Конструкция DO служит для распределения витков цикла между нитями, конструкция SECTIONS – для распределения между нитями указанных секций программы, а конструкция SINGLE указывает секцию, которая должна быть выполнена только одной нитью. При выходе из параллельной конструкции все порожденные на входе в нее нити сливаются с главной нитью, которая и продолжает дальнейшее выполнение.

В программе может быть произвольное число параллельных областей, причем допускается их вложенность. Для параллельной области можно указать классы используемых в ней переменных (общие или приватные). Имеются директивы высокоуровневой синхронизации (критические секции, барьер, и пр.).

Набор функций системы поддержки и переменных окружения служит для управления количеством создаваемых нитей, способами распределения между ними витков циклов, для низкоуровневой синхронизации нитей с помощью замков.

Подход OpenMP является диаметрально противоположным к подходу HPF:

- Вместо параллелизма по данным – параллелизм по управлению;
- Вместо статического анализа для автоматического поиска операторов, способных выполняться параллельно, – явное и полное задание параллелизма программистом;
- Вместо языка, требующего специального HPF-компилятора даже для работы на последовательной ЭВМ, – язык, позволяющий на последовательной ЭВМ компилироваться и выполняться в стандартной среде языка Фортран.

В настоящее время специалисты отмечают объединение подходов OpenMP и MPI.

Успешное внедрение OpenMP на мультипроцессорах и DSM–мультикомпьютерах резко активизировало исследования, направленные на поиски путей распространения OpenMP на мультикомпьютеры, кластеры и компьютерные сети. Указанные исследования сосредоточились, в основном, на двух направлениях:

- 1) Расширение языка средствами описания распределения данных;

2) Программная реализация системы DSM, использующей дополнительные указания компилятора, вставляемые им в выполняемую программу.

Первое направление представляется гораздо более перспективным для кластеров и компьютерных сетей, однако появление в ближайшие годы стандарта нового языка (расширенного OpenMP) не прогнозируется. В связи с этим все шире начинает использоваться гибридный подход, когда программа представляет собой систему взаимодействующих MPI-процессов, а каждый процесс программируется на OpenMP.

Подобный подход имеет преимущества с точки зрения упрощения программирования в том случае, когда в программе есть два уровня параллелизма – параллелизм между подзадачами и параллелизм внутри подзадачи. Такая ситуация возникает, например, при использовании многообластных (многоблочных) методов решения вычислительных задач.

Программировать на MPI сами подзадачи гораздо сложнее, чем их взаимодействие, поскольку распараллеливание подзадач связано с распределением элементов массивов и витков циклов между процессами. Организация же взаимодействия подзадач таких сложностей не вызывает, поскольку сводится к обмену между ними граничными значениями.

Широкое распространение SMP-кластеров также подталкивает к использованию гибридного подхода, поскольку использование OpenMP на мультипроцессоре может для некоторых задач дать заметный выигрыш в эффективности.

Основной недостаток этого подхода также очевиден – программисту надо знать и уметь использовать две разные модели параллелизма и разные инструментальные средства.

#### **1.2.4. Модель параллелизма по данным и управлению**

Модель, положенная в основу языков параллельного программирования Fortran-DVM и C-DVM, объединяет достоинства модели параллелизма по данным и модели параллелизма по управлению. Базирующаяся на этих языках система разработки параллельных программ (DVM) создана в Институте прикладной математики им. М.В. Келдыша Российской академии наук при активном участии студентов и аспирантов факультета вычислительной математики и кибернетики Московского государственного университета им. М.В. Ломоносова.

В отличие от модели параллелизма по данным, в системе DVM программист распределяет по процессорам виртуальной параллельной машины не только данные, но и соответствующие вычисления. При этом на него возлагается ответственность за соблюдение правила собственных вычислений. Кроме того, программист определяет общие данные, т.е. данные, вычисляемые на одних процессорах и используемые на других процессорах. И, наконец, он отмечает точки в последовательной программе, где происходит обновление значений общих данных.

При построении системы DVM, разработанной в 1994 году [4], использован новый подход, который характеризуется следующими принципами.

- 1) Система должна базироваться на высокоуровневой модели выполнения параллельной программы, удобной и понятной для программиста, программирующего на последовательных языках. Языки параллельного программирования должны представлять собой стандартные языки последовательного программирования, расширенные спецификациями параллелизма. Эти языки должны предлагать программисту модель программирования, достаточно близкую к модели выполнения. Знание программистом модели выполнения его программы и ее близость к модели программирования существенно упрощает для него анализ производительности программы и проведение ее модификаций, направленных на достижение приемлемой эффективности.
- 2) Спецификации параллелизма должны быть прозрачными для обычных компиляторов (например, оформляться в виде специальных комментариев). Во-первых, это упрощает внедрение новых параллельных языков, поскольку программист знает, что его программа без каких-либо изменений может выполняться в последовательном режиме на любых ЭВМ. Во-вторых, это позволяет использовать следующий метод поэтапной отладки DVM-программ. На первом этапе программа проходит отладку на рабочей станции как последовательная программа, с использованием обычных методов и средств отладки. На втором этапе программа выполняется на той же рабочей станции в специальном режиме проверки DVM-указаний. На третьем этапе программа может быть выполнена в специальном режиме, когда промежуточные результаты параллельного выполнения сравниваются с эталонными результатами (например, результатами последовательного выполнения).

3) Основная работа по реализации модели выполнения параллельной программы (например, распределение данных и вычислений) должна осуществляться динамически специальной системой – системой поддержки выполнения DVM-программ. Это позволяет обеспечить динамическую настройку DVM-программ при запуске (без перекомпиляции) на конфигурацию параллельного компьютера (количество процессоров, их производительность, латентность и пропускную способность коммуникационных каналов). Тем самым программист получает возможность иметь один вариант программы для выполнения на последовательных ЭВМ и параллельных ЭВМ различной конфигурации. Кроме того, на основании информации о выполнении DVM-программы на однопроцессорной ЭВМ можно посредством моделирования работы системы поддержки предсказать характеристики выполнения этой программы на параллельной ЭВМ с заданными параметрами (производительностью процессоров и коммуникационных каналов).

Проведем анализ основных возможностей языков Fortran–DVM и C–DVM. Программа на языках Fortran-DVM и C-DVM, помимо описания алгоритма обычными средствами языков Фортран 77 или С, содержит правила параллельного выполнения этого алгоритма.

Программисту предоставляются следующие возможности спецификации параллельного выполнения программы:

- Распределение элементов массива между процессорами;
- Распределение витков цикла между процессорами;
- Спецификация параллельно выполняющихся секций программы (параллельных задач) и отображение их на процессоры;
- Организация эффективного доступа к удаленным данным, расположенным на других процессорах;
- Организация эффективного выполнения глобальных операций с расположенными на различных процессорах данными (таких, как их суммирование или нахождение максимального или минимального значения).

Модель выполнения программы упрощенно описывается следующим образом. Параллельная программа на исходном языке Fortran–DVM (или C–DVM) превращается в программу на языке Фортран 77 (или С), содержащую вызовы функций системы поддержки и выполняющуюся в соответствии с моделью SPMD (одна программа – много данных) на каждом выделенном задаче процессоре. В момент запуска программы

существует единственная её ветвь (поток управления), которая начинает свое выполнение с первого оператора программы сразу на всех процессорах многопроцессорной системы.

Многопроцессорной системой (или системой процессоров) называется та машина, которая предоставляется программе пользователя аппаратурой и базовым системным программным обеспечением. Для распределённой ЭВМ примером такой машины может служить MPI-машина. В этом случае, многопроцессорная система – это группа MPI-процессов, которые создаются при запуске параллельной программы на выполнение.

Число процессоров многопроцессорной системы и её представление в виде многомерной решетки задаётся в командной строке при запуске программы. Все объявленные в программе переменные (за исключением специально указанных «распределённых» массивов) размножаются по всем процессорам.

При входе в параллельную конструкцию (параллельный цикл или область параллельных задач) ветвь разбивается на некоторое количество параллельных ветвей, каждая из которых выполняется на выделенном ей процессоре многопроцессорной системы.

При выходе из параллельной конструкции все ветви сливаются в ту же самую ветвь, которая выполнялась до входа в параллельную конструкцию.

Недостатком системы DVM является то, что предоставляются только параллельные расширения языков Фортран 77 и С, а расширения языков Фортран 90/95 и С++ отсутствуют. Правда, следует сказать, что Fortran-DVM базируется на расширенном языке Фортран 77, уже включающем в себя ряд возможностей Фортрана 90, и планируется дальнейшее такое расширение.

### **1.3. Сопоставление возможностей стандартных подходов к разработке и сопровождению параллельных программ**

Известны сравнительные характеристики Института прикладной математики им. М.В. Келдыша Российской академии наук по следующим вопросам [15]:

- Адекватна ли модель и язык программирования рассматриваемому классу задач?
- Какие методы и средства отладки предоставляются программистам?

- Сколько вариантов программы приходится сопровождать программисту?

Далее приводятся их основные аспекты.

### 1.3.1. Адекватность модели и языка программирования

Для класса задач, связанных с проведением научно-инженерных расчетов, вполне адекватен язык Фортран.

На достаточно характерном вычислительном алгоритме, реализующем метод релаксации Якоби для решения систем линейных уравнений, сравниваются размеры его последовательной версии и трех параллельных версий (MPI, HPF, DVM) [15].

Версия алгоритма с использованием гибридного подхода OpenMP+MPI не рассматривалась, поскольку невозможно применить возможности OpenMP для упрощения реализации данного алгоритма.

Размер в строках последовательной программы на языке Fortran 77 равен 17, а размеры параллельных версий для MPI, HPF и DVM равны соответственно 55, 24 и 21.

Такое же сравнение проводится на широко известных тестах NAS (NPB 2.3) [17]. Эти тесты хорошо отражают характер вычислительных задач различных классов.

Ниже в табл. 1.1 дается краткая характеристика тестов, и приводятся их размеры в строках для трех версий каждой программы – последовательной версии, MPI-версии и DVM-версии.

Информации о размерах программ с использованием OpenMP+MPI нет, но точно утверждается, что эти размеры лишь немного превышают размеры MPI-версий.

Информации о размерах HPF-версий также нет, но предполагается, что они незначительно отличаются от размеров DVM-версий.

Таблица 1.1.

Результаты тестов

Тест	Характеристика теста	SEQ	MPI	DVM	MPI/ SEQ	DVM/ SEQ
BT	3D Нильс-Стокс, метод переменных коэффициентов	3929	5744	3991	1.46	1.02
CG	Оценки наибольшего собственного значения симметрической положительной матрицы	1108	1793	1118	1.62	1.01
EP	Генерация пар случайных чисел Гусса	641	670	649	1.04	1.01
FT	Быстро преобразование Фурье, 3D	1500	2352	1603	1.57	1.07
IS	Проверка правильности	925	1218	1067	1.32	1.17
LU	3D Нильс-Стокс, метод второй релаксации	4189	5497	4269	1.31	1.02
MG	3D уравнение Пуассона, метод Multigrid	1898	2857	2131	1.50	1.12
SP	3D Нильс-Стокс, Bemn-Warming аргументы фиксирован	3361	5020	3630	1.49	1.08
I		17551	25151	18460	1.43	1.05

Смысл условных обозначений, указанных в табл. 1.1, заключается в следующем:

SEQ – последовательная программа;

MPI – параллельная программа с использованием Fortran 77+MPI или C+MPI;

DVM – параллельная программа на языке Fortran-DVM или C-DVM.

В качестве грубой оценки сложности программирования используются данные о соотношении количества дополнительных операторов, которые пришлось при распараллеливании тестов NAS добавить в их последовательные версии – 43% для MPI и 5% для DVM. Следует отметить при этом, что дополнительные операторы DVM-программы являются специальными комментариями, не зависящими от размеров массивов и числа процессоров. Дополнительный код MPI-программы представляет собой сложную систему программ управления передачей сообщений, зависящих от размеров массивов и числа процессоров.

Таким образом, лидируя по степени распространенности, параллельные версии программ с применением MPI уступают по сложности программирования параллельным версиям на базе DVM.

### 1.3.2. Методы и средства отладки

Отладка параллельной программы является процессом более трудоемким, чем отладка последовательной программы. Причиной этого является не только сложность параллельной программы, но и ее недетерминированное поведение, серьезно затрудняющее и функциональную отладку (достижение правильности результатов), и отладку эффективности программы. Раньше с подобными трудностями сталкивались, в основном, разработчики операционных систем и систем реального времени, которые сами и создавали для себя специальные средства отладки. Развитые средства отладки могут существенно упростить разработку параллельных программ прикладными программистами.

Большинство современных средств отладки параллельных программ основано на представлении программы как совокупности выполняющихся процессов. Средства функциональной отладки, как правило, предоставляют тот же набор примитивов, что и обычные

последовательные отладчики, расширенный с учетом специфики параллельного выполнения. Сюда входят следующие базовые примитивы:

- Инициализация выполнения программы;
- Завершение программы;
- Приостановка и продолжение выполнения программы;
- Задание точки останова, проверка заданных условий, просмотр и модификация значений переменных;
- Пошаговое исполнение.

Кроме того, широко используются средства накопления и анализа трассировки.

Наиболее развитыми системами функциональной отладки MPI-программ являются TotalView [18] и PGDBG [19].

При отладке эффективности используются разного рода профилировщики, выдающие после завершения программы информацию о временах вычислений, обменов сообщениями и синхронизации. Примерами таких систем, используемых для анализа эффективности MPI-программ, являются: Nupshot [20], Pablo [21], Vampir [22].

Несмотря на обилие различных инструментов для отладки MPI-программ положение дел в этой области нельзя признать удовлетворительным по следующим причинам [15].

Во-первых, каждая система отладки предоставляет свой набор возможностей и свой интерфейс с пользователем.

Во-вторых, каждая достаточно развитая система ориентирована на работу с конкретными компиляторами и библиотеками MPI. В результате, при переходе пользователя на другую машину, вероятнее всего, он не найдет там привычной для него системы отладки.

Некоторые из систем отладки MPI-программ адаптированы для отладки HPF-программ, а также программ, использующих гибридный подход OpenMP+MPI. Для функциональной отладки DVM-программ предложена и реализована следующая методика поэтапной отладки программ. На первом этапе программа отлаживается на рабочей станции как последовательная программа, используя обычные методы и средства отладки. На втором этапе программа выполняется на той же рабочей станции в специальном режиме моделирования параллельного выполнения для проверки корректности распараллеливающих указаний. На третьем этапе программа может быть выполнена на параллельной машине в специальном режиме, когда промежуточные результаты параллельного выполнения сравниваются с эталонными результатами

(например, результатами последовательного выполнения). Для обеспечения второго и третьего этапов этой методики служит специальный DVM-отладчик. Для отладки эффективности DVM-программ используется анализатор производительности, который позволяет пользователю получить информацию об основных характеристиках эффективности выполнения его программы (или ее частей) на параллельной системе.

Для облегчения отладки эффективности можно использовать специальный инструмент-предиктор, позволяющий на рабочей станции смоделировать выполнение DVM-программы на параллельной ЭВМ с заданными параметрами (топологии коммуникационной сети, ее пропускной способности, а также производительности процессоров) и получить прогнозируемые характеристики эффективности ее выполнения.

### 1.3.3. Сопровождаемые варианты программы

Если параллельная программа разработана с использованием MPI, то использовать ее на обычных персональных компьютерах или рабочих станциях затруднительно по двум основным причинам:

- Для ее компиляции и выполнения требуется наличие библиотеки MPI;
- Программа должна быть написана так, чтобы она могла выполняться на одном процессоре.

Поэтому, как правило, программист имеет и вынужден сопровождать два варианта программы – для последовательного и для параллельного выполнения.

Программа на языке HPF способна выполнятся на одном процессоре. Однако использовать ее на обычных персональных компьютерах или рабочих станциях невозможно, если там отсутствует компилятор с языка HPF.

Поскольку DVM-указания прозрачны для обычных компиляторов, то один вариант DVM-программы может использоваться и для параллельного выполнения, и для последовательного выполнения на персональных компьютерах или рабочих станциях.

## 1.4. Эффективность выполнения параллельных программ по тестам NBP

Эффективность выполнения программ всегда являлась очень важным фактором, определявшим в значительной степени успех и

распространение языков программирования, предназначенных для создания вычислительных программ.

В настоящее время, когда распараллеливание программы ускоряет ее выполнение в сотни раз, а трудоемкость создания параллельных программ является основным препятствием для широкого использования высокой производительности современных вычислительных систем, эффективность выполнения программ, тем не менее, по-прежнему остается важным фактором при выборе программистом того или иного подхода для разработки параллельных программ.

Данные тестов из пакета NPB, приводимых Институтом прикладной математики им. М.В. Келдыша Российской академии наук [23, 24], позволяют сделать следующие выводы:

1. MPI-программы, как правило, выполняются быстрее остальных. Разрыв увеличивается по мере роста числа процессоров.
2. При использовании подхода OpenMP+MPI в MPI-программу для отдельного узла вставлялись директивы OpenMP. Даже на мультипроцессоре с 4 процессорами на общей памяти, используемом в качестве узла IBM SP, OpenMP проигрывает на регулярных программах MPI-подходу. Это объясняется лучшей локализацией доступа к данным в MPI-программе, что приводит к более эффективному использованию кэш-памяти.
3. Большой проигрыш HPF при увеличении числа процессоров объясняется принципиальными недостатками HPF-подхода, на которые указывалось выше.
4. Эффективность DVM-программ гораздо выше, чем эффективность HPF, и вполне сравнима с эффективностью MPI. Для оценки масштабируемости DVM-программ приведены данные о соотношении времен выполнения MPI-версий и DVM-версий на ЭВМ MBC-1000M [25] для всех восьми тестов класса С (тесты этого класса ориентированы на высокопроизводительные параллельные системы, требуют много памяти и поэтому не могут выполняться на малом числе процессоров). В среднем эффективность DVM-версий составляет около 80% от эффективности MPI-версий.

## **1.5. Переносимость и повторное использование параллельных программ**

В настоящее время, когда программист имеет возможность запускать свою программу на разных, порою географически удаленных

параллельных системах, очень важна переносимость программ (их способность выполнятся на различных вычислительных системах с приемлемой эффективностью).

Создать для новых параллельных систем прикладное программное обеспечение, необходимое для решения важнейших научно-технических задач, вряд ли возможно без повторного использования уже созданных программ, без накопления и использования богатых библиотек параллельных программ.

Поэтому переносимость программ и их способность к повторному использованию должны рассматриваться как самые первостепенные факторы качества параллельных программ.

Как уже указывалось выше, широкое внедрение MPI обеспечило переносимость программ, разрабатываемых в рамках MPI-подхода для однородных вычислительных систем. Определенные проблемы возникают из-за различий в организации внешней памяти на разных системах. Например, для эффективной работы с локальными дисками, имеющимися на каждом узле системы, программа должна соответствующим образом это организовать. Переход на систему, на котором используется единый файл-сервер и нет локальных дисков, потребует изменения такой программы. Использовать же параллельный ввод-вывод, предложенный в стандарте MPI-2, тяжело из-за его сложности и из-за того, что он поддерживается не на всех системах.

Гораздо хуже обстоят дела с повторным использованием MPI-программ. Очень сложно разрабатывать программы, способные выполнятся на различном числе процессоров с разными по объему данными. Многие программисты предпочитают иметь разные варианты программ для разных конфигураций данных и процессоров, а также иметь отдельный вариант программы для работы на однопроцессорной ЭВМ. В таких условиях использование чужой программы представляется гораздо менее вероятной, чем это было на традиционных ЭВМ.

Но главные трудности связаны с отсутствием в языке программирования такого понятия, как распределенный массив. Вместо такого единого массива в программе используется на каждом процессоре свой локальный массив. Их соответствие исходному массиву, с которым программа имела бы дело при ее выполнении на одном процессоре, зафиксировано только в голове программиста и, возможно, в комментариях к программе.

Отсутствие в языке понятия распределенного массива является серьезным препятствием для разработки и использования библиотек стандартных параллельных программ. В результате, каждая такая библиотека вынуждена вводить свое понятие распределенного массива и реализовывать свой набор операций над ним. Однако вызов таких стандартных программ из прикладной программы требует от программиста согласования разных представлений распределенных массивов.

Все вышесказанное в значительной мере относится и к гибридному подходу OpenMP+MPI.

Таких принципиальных проблем с повторным использованием параллельных программ нет, если эти программы разрабатываются в рамках подходов HPF или DVM. Более того, в рамках этих подходов можно обеспечить удобный вызов стандартных параллельных программ, разработанных на других языках и входящих в состав известных библиотек (для которых известно их внутреннее представление распределенных массивов). Например, можно позволить вызывать функции пакета ScaLAPACK [26] из HPF-программ или DVM-программ. Переносимость HPF-программ определяется наличием компилятора HPF для конкретной параллельной системы, а точнее для процессора, используемого в узлах этой системы. Поскольку качественных и свободно распространяемых компиляторов, способных генерировать программы для всех процессоров, нет, то при переносе HPF-программ на некоторые платформы возникают серьезные проблемы.

Переносимость DVM-программы на однородные вычислительные системы обеспечивается тем, что она преобразуется в программу на языке Фортран 77 (или С), содержащую вызовы функций системы поддержки, которая для организации межпроцессорного взаимодействия использует библиотеку MPI. Такая программа может выполняться всюду, где есть MPI и компиляторы с языками С и Фортран 77. Система поддержки может учсть особенности организации внешней памяти и обеспечить ее эффективное использование, не требуя изменений в DVM-программе.

Кроме того, программы на языке Fortran-DVM могут автоматически конвертироваться в программы на языке HPF, HPF2 и OpenMP.

Способность параллельных программ эффективно выполняться на неоднородных вычислительных системах и сетях ЭВМ требует отдельного рассмотрения.

Все три подхода, базирующиеся на существующих стандартах (MPI, HPF, OpenMP+MPI), не рассчитаны на применение в неоднородных распределенных системах. Поэтому перенос таких программ на неоднородные системы и сети ЭВМ приведет к заметной потере их эффективности, поскольку прикладной программист фактически не имел возможностей для написания программ, способных настраиваться на различную производительность процессоров и коммуникационных каналов.

В DVM-системе реализованы возможности задания производительностей процессоров (или их автоматического определения при запуске программы) и их учета при распределении данных и вычислений между процессорами. Это позволяет DVM-программам эффективно выполняться на неоднородных системах. Тесты NAS на модели неоднородного кластера, приводимые Институтом прикладной математики им. М.В. Келдыша Российской академии наук [15] показали, что время выполнения MPI и DVM версий тестов NAS (класс С) на неоднородном кластере ниже у MPI-программ.

В результате сопоставления всех приведенных выше данных можно сделать вывод, что библиотека MPI является одним из ведущих средств программирования параллельных вычислений.

## 1.6. Выводы

Сравнительный анализ современных подходов к разработке и исследованию программного обеспечения для высокопроизводительных вычислительных систем выявил следующие особенности:

- Основное внимание уделяется сопоставлению моделей параллельного программирования, среди которых модель передачи сообщений, модель параллелизма по данным, гибридная модель по управлению с передачей сообщений, модель параллелизма по данным и управлению;
- В рамках выбираемой модели параллельного программирования процесс определения качества функционирования программы отчужден от ее выполнения;
- Качество функционирования разработанных программ выясняется при тестировании;
- При тестировании ориентируются на отдельные спецификации функциональных задач вычислительной математики или информационного обмена при вариациях в масштабируемости;

- При основном приоритете снижения временных затрат на выполнение предпочтение отдается модели передачи сообщений;
- При модификации существующих программ и создании новых версий не учитываются в их реализациях современные возможности аналитических методов определения качества функционирования.

Наряду с этим, результаты научных исследований по определению и исследованию динамических характеристик параллельных программ [32, 33, 34] показывают перспективность их использования в реализациях программных систем, поскольку позволяют осуществить оперативный мониторинг качества функционирования программного обеспечения высокопроизводительных вычислительных систем.

В данном пособии раскрывается новый путь развития современных подходов к разработке программного обеспечения для высокопроизводительных вычислительных систем в классе моделей передачи сообщений параллельно выполняемых программ на основе интеллектуализации процессов оценки их динамических характеристик.

Предусматриваемая интеллектуализация ориентирована на включение в функциональную спецификацию параллельных программ компонент, предназначенных для оценки динамических характеристик.

В качестве прикладного объекта выбирается программный компонент систем третьего класса защищенности, функциональная спецификация которых является опорной для систем более высоких классов защищенности. Представленный выбор обусловлен актуальностью совершенствования программного обеспечения систем информационной безопасности.

В пособии описывается процесс создания научно-обоснованной системы прототипов ключевых программных компонентов в базисе функций библиотеки MPI для определения качества работы программного обеспечения высокопроизводительных вычислительных систем. При создании указанной системы решаются следующие задачи:

1. Определение характеристик спецификаций базиса функций библиотеки MPI и детализация ключевых особенностей их применения;
2. Построение прототипов ключевых компонентов программного обеспечения в базисе функций библиотеки MPI для оценки статистических характеристик времени выполнения параллельных программ;
3. Анализ работоспособности ключевых компонентов программного обеспечения;

4. Формирование прототипа программы для оценки статистических характеристик обнаружения и отражения угроз в автоматизированных системах третьего класса защищенности в базисе функций MPI.

## **2. ХАРАКТЕРИСТИКА СПЕЦИФИКАЦИИ БАЗИСА ФУНКЦИЙ БИБЛИОТЕКИ MPI**

Коммуникационная библиотека MPI стала общепризнанным стандартом в параллельном программировании с использованием механизма передачи сообщений. Полное и строгое описание среды программирования MPI можно найти в авторском описании разработчиков на английском языке [7, 8]. Более подробные описания функций стандарта, в переводе на русский язык приводятся в Приложении 1. Далее раскрыты общие сведения и названия функций, которые необходимы для разработки требуемого программного обеспечения.

### **2.1. Общая организация библиотеки MPI**

MPI-программа представляет собой набор независимых процессов, каждый из которых выполняет свою собственную программу, написанную на языке C или FORTRAN. Появились реализации MPI для C++, однако разработчики стандарта MPI за них ответственности не несут. Процессы MPI-программы взаимодействуют друг с другом посредством вызова коммуникационных процедур. Как правило, каждый процесс выполняется в своем собственном адресном пространстве, однако допускается и режим разделения памяти.

MPI не специфицирует модель выполнения процесса – это может быть как последовательный процесс, так и многопотоковый. MPI не предоставляет никаких средств для распределения процессов по вычислительным узлам и для запуска их на исполнение. Эти функции возлагаются либо на операционную систему, либо на программиста. MPI не накладывает каких-либо ограничений на то, как процессы будут распределены по процессорам, в частности, возможен запуск MPI-программы с несколькими процессами на обычной однопроцессорной системе.

Для идентификации наборов процессов вводится понятие *группы*, объединяющей все или какую-то часть процессов. Каждая группа образует *область связи*, с которой связывается специальный объект – *коммуникатор* области связи. Процессы внутри группы нумеруются целым числом в диапазоне  $0..groupsize-1$ . Все коммуникационные операции с некоторым коммуникатором будут выполняться только внутри области связи, описываемой этим коммуникатором. При инициализации

MPI создается предопределенная область связи, содержащая все процессы MPI-программы, с которой связывается предопределенный коммуникатор MPI\_COMM\_WORLD. В большинстве случаев на каждом процессоре запускается один отдельный процесс, и тогда термины процесс и процессор становятся синонимами, а величина group size становится равной NPROCS – числу процессоров, выделенных задаче.

MPI достаточно объемная и сложная библиотека, состоящая примерно из 130 функций, в число которых входят:

- функции инициализации и закрытия MPI-процессов;
- функции, реализующие коммуникационные операции типа точка-точка;
- функции, реализующие коллективные операции;
- функции для работы с группами процессов и коммуникаторами;
- функции для работы со структурами данных;
- функции формирования топологии процессов.

В принципе, любая параллельная программа может быть написана с использованием всего 6 MPI-функций, а достаточно полную и удобную среду программирования составляет набор из 24 функций [35].

Каждая из MPI-функций характеризуется способом выполнения:

1. *Локальная функция* — выполняется внутри вызывающего процесса. Ее завершение не требует коммуникаций.

2. *Нелокальная функция* — для ее завершения требуется выполнение MPI-процедуры другим процессом.

3. *Глобальная функция* — процедуру должны выполнять все процессы группы. Несоблюдение этого условия может приводить к зависанию задачи.

4. *Блокирующая функция* — возврат управления из процедуры гарантирует возможность повторного использования параметров, участвующих в вызове. Никаких изменений в состоянии процесса, вызвавшего блокирующий запрос, до выхода из процедуры не может происходить.

5. *Неблокирующая функция* — возврат из процедуры происходит немедленно, без ожидания окончания операции и до того, как будет разрешено повторное использование параметров, участвующих в запросе. Завершение неблокирующих операций осуществляется специальными функциями.

В языке С все процедуры являются функциями, большинство из них возвращает код ошибки. При использовании имен подпрограмм и именованных констант необходимо строго соблюдать регистр символов. Массивы индексируются с 0. Логические переменные представляются типом int (true соответствует 1, а false – 0). Определение всех именованных констант, прототипов функций и определение типов выполняется подключением файла mpi.h. Введение собственных типов в MPI было продиктовано тем обстоятельством, что стандартные типы языков на разных платформах имеют различное представление. MPI допускает возможность запуска процессов параллельной программы на компьютерах различных платформ, обеспечивая при этом автоматическое преобразование данных при пересылках.

## 2.2. Базовые функции библиотеки MPI

Любая прикладная MPI-программа должна начинаться с вызова функции инициализации MPI-функции MPI\_Init. В результате выполнения этой функции создается группа процессов, в которую помещаются все процессы приложения, и создается область связи, описываемая предопределенным коммуникатором MPI\_COMM\_WORLD. Эта область связи объединяет все процессы-приложения. Процессы в группе упорядочены и пронумерованы от 0 до groupszie–1, где groupszie равно числу процессов в группе. Кроме этого, создается предопределенный коммуникатор MPI\_COMM\_SELF, описывающий свою область связи для каждого отдельного процесса.

Синтаксис функции инициализации MPI\_Init:

```
int MPI_Init(int *argc, char ***argv)
```

Функция завершения MPI-программ MPIFinalize:

```
int MPI_Finalize(void)
```

Функция определения числа процессов в области связи

*MPI\_Comm\_size*:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Функция определения номера процесса MPI\_Comm\_rank:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

В минимальный набор следует включить также две функции передачи и приема сообщений.

*Функция передачи сообщения MPI\_Send:*

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

*Функция приема сообщения MPI\_Recv:*

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Для определения временных параметров работы программы потребуются следующие две функции:

*Функция отсчета времени (таймер) MPI\_Wtime:*

```
double MPI_Wtime(void)
```

*Функция MPI\_Wtick*, имеющая точно такой же синтаксис, возвращает разрешение таймера (минимальное значение кванта времени).

## 2.3. Коммуникационные операции типа точка-точка

### 2.3.1. Обзор коммуникационных операций типа точка-точка

К операциям этого типа относятся две представленные в предыдущем разделе коммуникационные процедуры. В коммуникационных операциях типа точка-точка всегда участвуют не более двух процессов: передающий и принимающий. В MPI имеется множество функций, реализующих такой тип обменов. Многообразие объясняется возможностью организации таких обменов множеством способов. Описанные в предыдущем разделе функции реализуют *стандартный режим с блокировкой*.

*Блокирующие функции* подразумевают выход из них только после полного окончания операции, т.е. вызывающий процесс блокируется, пока операция не будет завершена. Для функции посылки сообщения это означает, что все пересылаемые данные помещены в буфер (для разных реализаций MPI это может быть либо какой-то промежуточный системный буфер, либо непосредственно буфер получателя). Для функции приема сообщения блокируется выполнение других операций, пока все данные из буфера не будут помещены в адресное пространство принимающего процесса.

*Неблокирующие функции* подразумевают совмещение операций обмена с другими операциями, поэтому неблокирующие функции передачи и приема по сути дела являются функциями инициализации соответствующих операций. Для опроса завершенности операции (и завершения) вводятся дополнительные функции. Как для блокирующих, так и неблокирующих операций MPI поддерживает четыре режима выполнения. Эти режимы касаются только функций передачи данных, поэтому для блокирующих и неблокирующих операций имеется по четыре функции посылки сообщения. В табл. 2.1 перечислены имена базовых коммуникационных функций типа точка-точка, имеющихся в библиотеке MPI.

Таблица 2.1.  
Список коммуникационных функций типа точка-точка

Режимы	С блокировкой	Без блокировки
Стандартная посылка	MPI Send	MPI Isend
Синхронная посылка	MPI Ssend	MPI Issend
Буферизованная	MPI Bsend	MPI Ibsend
Согласованная	MPI Rsend	MPI Irsend
Прием информации	MPI Recv	MPI Irecv

Содержание выше представленной таблицы иллюстрирует принцип формирования имен функций. К именам базовых функций Send/Recv добавляются различные префиксы. Префикс S (synchronous) – означает синхронный режим передачи данных. Операция передачи данных заканчивается только тогда, когда заканчивается прием данных. Функция с префиксом S считается нелокальной. Префикс B (buffered) – означает буферизованный режим передачи данных. В адресном пространстве передающего процесса с помощью специальной функции создается буфер обмена, который используется в операциях обмена. Операция посылки заканчивается, когда данные помещены в этот буфер. Функция имеет локальный характер.

Префикс R (ready) представляет согласованный или подготовленный режим передачи данных. Операция передачи данных начинается только тогда, когда принимающий процессор выставил признак готовности приема данных, инициировав операцию приема. Функция с префиксом R является нелокальной. Префикс I (immediate) относится к неблокирующим операциям. Все функции передачи и приема сообщений могут использоваться в любой комбинации друг с другом. Функции передачи, находящиеся в одном столбце, имеют

совершенно одинаковый синтаксис и отличаются только внутренней реализацией.

### 2.3.2.Блокирующие коммуникационные операции

Синтаксис базовых коммуникационных функций MPI\_Send и MPI\_Recv рассмотрен в подразделе 2.3.1, поэтому здесь приводится только семантика этих операций. В стандартном режиме выполнение операции обмена включает три этапа. Передающая сторона формирует пакет сообщения, в который помимо передаваемой информации упаковываются адрес отправителя (source), адрес получателя (dest), идентификатор сообщения (tag) и коммуникатор (comm). Этот пакет передается отправителем в системный буфер, и на этом функция посылки сообщения заканчивается. Сообщение системными средствами передается адресату. Принимающий процессор извлекает сообщение из системного буфера, когда у него появится потребность в этих данных. Содержательная часть сообщения помещается в адресное пространство принимающего процесса (параметр buf), а служебная в параметр status. Поскольку операция выполняется в асинхронном режиме, адресная часть принятого сообщения состоит из трех полей:

- коммуникатора (comm), поскольку каждый процесс может одновременно входить в несколько областей связи;
- номера отправителя в этой области связи (source);
- идентификатора сообщения (tag), который используется для взаимной привязки конкретной пары операций посылки и приема сообщений.

Параметр count (количество принимаемых элементов сообщения) в процедуре приема сообщения должен быть не меньше, чем длина принимаемого сообщения. При этом реально будет приниматься столько элементов, сколько находится в буфере. Подобная реализация операции чтения связана с тем, что MPI допускает использование расширенных запросов:

- для идентификаторов сообщений (MPI\_ANY\_TAG читать сообщение с любым идентификатором);
- для адресов отправителя (MPI\_ANY\_SOURCE читать сообщение от любого отправителя).

Не допускается расширенных запросов для коммуникаторов. Расширенные запросы возможны только в операциях чтения. В этом отражается фундаментальное свойство механизма передачи сообщений:

асимметрия операций передачи и приема сообщений, связанная с тем, что инициатива в организации обмена принадлежит передающей стороне. Таким образом, после чтения сообщения некоторые параметры могут оказаться неизвестными, а именно: число считанных элементов, идентификатор сообщения и адрес отправителя. Эту информацию можно получить с помощью параметра `status`. Переменные `status` должны быть явно объявлены в MPI-программе. В языке C `status` – это структура типа `MPI_Status` с тремя полями `MPI_SOURCE` (процесс отправитель), `MPI_TAG` (идентификатор сообщения), `MPI_ERROR` (код ошибки). Количество считанных элементов в переменную `status` не заносится.

Далее перечислены основные функции этого класса:

`MPI_Get_count`  
`int MPI_Probe`  
`MPI_Sendrecv`  
`MPI_Sendrecv_replace`

### 2.3.3. Неблокирующие коммуникационные операции

Использование неблокирующих коммуникационных операций повышает безопасность с точки зрения возникновения тупиковых ситуаций, а также может увеличить скорость работы программы за счет совмещения выполнения вычислительных и коммуникационных операций. Эти задачи решаются разделением коммуникационных операций на две стадии: инициирование операции и проверка завершения операции. Неблокирующие операции используют специальный скрытый (opaque) объект «запрос обмена» (`request`) для связи между функциями обмена и функциями опроса их завершения. Для прикладных программ доступ к этому объекту возможен только через вызовы MPI-функций. Если операция обмена завершена, подпрограмма проверки снимает «запрос обмена», устанавливая его в значение `MPI REQUEST NULL`. Снять запрос без ожидания завершения операции можно подпрограммой `MPI_Request_free`.

Основные функции этого типа:

`MPI_Isend`  
`MPI_Irecv`  
`MPI_Iprobe`

Имеется два типа функций завершения неблокирующих операций (ожидание завершения и проверка завершения):

1. Операции семейства `WAIT` блокируют работу процесса до полного

завершения операции.

2. Операции семейства TEST возвращают значения TRUE или FALSE в зависимости от того, завершилась операция или нет. Они не блокируют работу процесса и полезны для предварительного определения факта завершения операции.

В MPI имеется набор подпрограмм для одновременной проверки на завершение нескольких операций. Их перечень приведен в табл. 2.2. Кроме того, MPI позволяет для неблокирующих операций формировать целые пакеты запросов на коммуникационные операции MPI\_Send\_init и MPI\_Recv\_init, которые запускаются функциями MPI\_Start или MPI\_Startall. Проверка на завершение выполнения производится обычными средствами с помощью функций семейства WAIT и TEST.

Таблица 2.2.

Функции коллективного завершения неблокирующих операций

Выполняемая проверка	Функции ожидания (блокирующие)	Функции проверки (неблокирующие)
Завершились все операции	MPI_Waitall	MPI_Testall
Завершилась по крайней мере одна операция	MPI_Waitany	MPI_Testany
Завершилась одна из списка проверяемых	MPI_Waitsome	MPI_Testsome

## 2.4. Коллективные операции

### 2.4.1. Обзор коллективных операций

Набор операций типа точка-точка является достаточным для программирования любых алгоритмов, однако MPI вряд ли бы завоевал такую популярность, если бы ограничивался только этим набором коммуникационных операций. Одной из наиболее привлекательных сторон MPI является наличие широкого набора коллективных операций, которые берут на себя выполнение наиболее часто встречающихся при программировании действий. Например, часто возникает потребность разослать некоторую переменную или массив из одного процессора всем остальным. Каждый программист может написать такую процедуру с использованием операций Send/Recv, однако гораздо удобнее воспользоваться коллективной операцией MPI\_Bcast. Причем

гарантировано, что эта операция будет выполняться гораздо эффективнее, поскольку MPI-функция реализована с использованием внутренних возможностей коммуникационной среды. Главное отличие коллективных операций от операций типа точка-точка состоит в том, что в них всегда участвуют все процессы, связанные с некоторым коммуникатором. Несоблюдение этого правила приводит либо к аварийному завершению задачи, либо к еще более неприятному зависанию задачи.

Набор коллективных операций включает:

- Синхронизацию всех процессов с помощью барьеров(MPI\_Barrier).
- Коллективные коммуникационные операции, в число которых входят:
  - рассылка информации от одного процесса всем остальным членам некоторой области связи (MPI\_Bcast);
  - сборка (gather) распределенного по процессам массива в один массив с сохранением его в адресном пространстве выделенного (root) процесса (MPI\_Gather, MPI\_Gatherv);
  - сборка (gather) распределенного массива в один массив с рассылкой его всем процессам некоторой области связи (MPI\_Allgather, MPI\_Allgatherv);
  - разбиение массива и рассылка его фрагментов (scatter) всем процессам области связи (MPI\_Scatter, MPI\_Scatterv);
  - совмещенная операция Scatter/Gather (All-to-All), каждый процесс делит данные из своего буфера передачи и разбрасывает фрагменты всем остальным процессам, одновременно собирая фрагменты, посланные другими процессами, в свой буфер приема (MPI\_Alltoall, MPI\_Alltoallv).
- Глобальные вычислительные операции (sum, min, max и др.) над данными, расположенными в адресных пространствах различных процессов:
  - с сохранением результата в адресном пространстве одного процесса (MPI\_Reduce);
  - с рассылкой результата всем процессам (MPI\_Allreduce);
  - совмещенная операция Reduce/ Scatter (MPI\_Reduce\_scatter);
  - префиксная редукция (MPI\_Scan).

Все коммуникационные подпрограммы, за исключением `MPI_Bcast`, представлены в двух вариантах:

- простой вариант, когда все части передаваемого сообщения имеют одинаковую длину и занимают смежные области в адресном пространстве процессов;
- «векторный» вариант, который предоставляет более широкие возможности по организации коллективных коммуникаций, снимая ограничения, присущие простому варианту, как в части длин блоков, так и в части размещения данных в адресном пространстве процессов. Векторные варианты отличаются дополнительным символом «`v`» в конце имени функции.

Отличительные особенности коллективных операций:

- Коллективные коммуникации не взаимодействуют с коммуникациями типа точка-точка.
- Коллективные коммуникации выполняются в режиме с блокировкой. Возврат из подпрограммы в каждом процессе происходит тогда, когда его участие в коллективной операции завершилось, однако это не означает, что другие процессы завершили операцию.
- Количество получаемых данных должно быть равно количеству посланных данных.
- Типы элементов посылаемых и получаемых сообщений должны совпадать.
- Сообщения не имеют идентификаторов.

Функция синхронизации процессов `MPI_Barrier` блокирует работу вызвавшего ее процесса до тех пор, пока все другие процессы группы также не вызовут эту функцию. Завершение работы этой функции возможно только всеми процессами одновременно (все процессы «преодолевают барьер» одновременно).

#### 2.4.2. Функции сбора блоков данных от всех процессов группы

Семейство функций сбора блоков данных от всех процессов группы состоит из четырех подпрограмм: `MPI_Gather`, `MPI_Allgather`, `MPI_Gatherv`, `MPI_Allgatherv`. Каждая из указанных подпрограмм расширяет функциональные возможности предыдущих.

#### **2.4.3. Функции распределения блоков данных по всем процессам группы**

Семейство функций распределения блоков данных по всем процессам группы состоит из двух подпрограмм: MPI\_Scatter и MPI\_Scatterv.

#### **2.4.4. Совмещенные коллективные операции**

Функция *MPI\_Alltoall* совмещает в себе операции Scatter и Gather и является расширением операции Allgather, когда каждый процесс посыпает различные данные разным получателям. Процесс i посыпает j-ый блок своего буфера sendbuf процессу j, который помещает его в i-ый блок своего буфера recvbuf. Количество посланных данных должно быть равно количеству полученных данных для каждой пары процессов. Функция *MPI\_Alltoallv* реализует векторный вариант операции Alltoall, допускающий передачу и прием блоков различной длины с более гибким размещением передаваемых и принимаемых данных.

#### **2.4.5. Глобальные вычислительные операции над распределенными данными**

В параллельном программировании математические операции над блоками данных, распределенных по процессорам, называют *глобальными операциями редукции*. В общем случае *операцией редукции* называется операция, аргументом которой является вектор, а результатом – скалярная величина, полученная применением некоторой математической операции ко всем компонентам вектора. В частности, если компоненты вектора расположены в адресных пространствах процессов, выполняющихся на различных процессорах, то в этом случае отличают *глобальную (параллельную) редукцию*. Например, пусть в адресном пространстве всех процессов некоторой группы процессов имеются копии переменной var (необязательно имеющие одно и то же значение), тогда применение к ней операции вычисления глобальной суммы или, другими словами, операции редукции SUM возвратит одно значение, которое будет содержать сумму всех локальных значений этой переменной. В MPI глобальные операции редукции представлены в нескольких вариантах:

- с сохранением результата в адресном пространстве одного процесса (MPI\_Reduce);

- с сохранением результата в адресном пространстве всех процессов (MPI\_Allreduce);
- префиксная операция редукции, которая в качестве результата операции возвращает вектор. 1-я компонента этого вектора является результатом редукции первых  $i$  компонент распределенного вектора (MPI\_Scan);
- совмещенная операция Reduce/Scatter (MPI\_Reduce\_scatter).

Описание синтаксиса данных функций приведено в Приложении 1.

## 2.5. Производные типы данных и передача упакованных данных

Рассмотренные ранее коммуникационные операции позволяют посылать или получать последовательность элементов одного типа, занимающих смежные области памяти. При разработке параллельных программ иногда возникает потребность передавать данные разных типов (например, структуры) или данные, расположенные в несмежных областях памяти (части массивов, не образующих непрерывную последовательность элементов). MPI предоставляет два механизма эффективной пересылки данных в упомянутых выше случаях:

- путем создания производных типов для использования в коммуникационных операциях вместо предопределенных типов MPI;
- пересылку упакованных данных (процесс-отправитель упаковывает пересылаемые данные перед их отправкой, а процесс-получатель распаковывает их после получения).

В большинстве случаев оба эти механизма позволяют добиться желаемого результата, но в конкретных случаях более эффективным может оказаться либо один, либо другой подход.

### 2.5.1. Производные типы данных

Производные типы MPI не являются в полном смысле типами данных, как это понимается в языках программирования. Они не могут использоваться ни в каких других операциях, кроме коммуникационных. Производные типы MPI следует понимать как описатели расположения в памяти элементов базовых типов. Производный тип MPI представляет собой скрытый (opaque) объект, который специфицирует две последовательности: последовательность базовых типов и последовательность смещений. Последовательность таких пар определяется как отображение (карта) типа:

Типетар =  $\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$

Значения смещений не обязательно должны быть неотрицательными, различными и упорядоченными по возрастанию. Отображение типа вместе с базовым адресом начала расположения данных buf определяет коммуникационный буфер обмена. Этот буфер будет содержать  $n$  элементов, а  $i$ -й элемент будет иметь адрес  $buf + desp$  и иметь базовый тип type. Стандартные типы MPI имеют предопределенные отображения типов. Например, MPI\_INT имеет отображение  $\{(int, 0)\}$ .

Использование производного типа в функциях обмена сообщениями можно рассматривать как трафарет, наложенный на область памяти, которая содержит передаваемое или принятое сообщение.

Стандартный сценарий определения и использования производных типов включает следующие шаги:

- Производный тип строится из предопределенных типов MPI и ранее определенных производных типов с помощью специальных функций-конструкторов

`MPI_Type_contiguous, MPI_Type_vector, MPI_Type_hvector,  
MPI_Type_indexed, MPI_Type_hindexed, MPI_Type_struct.`

• Новый производный тип регистрируется вызовом функции `MPI_Type_commit`. Только после регистрации новый производный тип можно использовать в коммуникационных подпрограммах и при конструировании других типов. Предопределенные типы MPI считаются зарегистрированными.

- Когда производный тип становится ненужным, он уничтожается функцией `MPI_Type_free`.

Любой тип данных в MPI имеет две характеристики: протяженность и размер, выраженные в байтах:

• Протяженность типа определяет, сколько байт переменная данного типа занимает в памяти. Эта величина может быть вычислена как адрес последней ячейки данных – адрес первой ячейки данных + длина последней ячейки данных (опрашивается подпрограммой `MPI_Type_extent`).

• Размер типа определяет количество реально передаваемых байт в коммуникационных операциях. Эта величина равна сумме длин всех базовых элементов определяемого типа (опрашивается подпрограммой `MPI_Type_size`).

Для простых типов протяженность и размер совпадают.

### 2.5.2. Передача упакованных данных

Для посылки элементов разного типа из нескольких областей памяти их следует предварительно запаковать в один массив, последовательно обращаясь к функции упаковки MPI\_Pack. При первом вызове функции упаковки параметр position, как правило, устанавливается в 0, чтобы упакованное представление размещалось с начала буфера. Для непрерывного заполнения буфера необходимо в каждом последующем вызове использовать значение параметра position, полученное из предыдущего вызова.

Упакованный буфер пересылается любыми коммуникационными операциями с указанием типа MPI\_PACKED и коммуникатора comm, который использовался при обращениях к функции MPI\_Pack.

Полученное упакованное сообщение распаковывается в различные массивы или переменные. Это реализуется последовательными вызовами функции распаковки MPI\_Unpack с указанием числа элементов, которое следует извлечь при каждом вызове, и с передачей значения position, возвращенного предыдущим вызовом. При первом вызове функции параметр position следует установить в 0. В общем случае, при первом обращении должно быть установлено то значение параметра position, которое было использовано при первом обращении к функции упаковки данных. Очевидно, что для правильной распаковки данных очередьность извлечения данных должна быть той же самой, как и при упаковке. Функция MPI\_Pack\_size помогает определить размер буфера, необходимый для упаковки некоторого количества данных типа datatype.

## 2.6. Сравнение версий библиотеки MPI-1.1 и MPI-2

В функциональности MPI-1.1 есть пробелы, которые устраниены в MPI-2. Спецификация на MPI-2 включает в себя следующие нововведения:

- Развит механизм взаимодействия между приложениями. Обеспечена поддержка механизма «клиент–сервер». В результате подобных расширений можно решать на MPI-2 не только расчетные математические задачи, но и задачи системы массового обслуживания (обслуживания базы данных и прочее).
- Обеспечено динамическое порождение ветвей. Для программирования расчетных задач это не нужно, однако такая возможность однозначно необходима для решения задач систем массового обслуживания.

- Создан архитектурно-независимый интерфейс для работы с файлами. Это имеет особенное значение, если диск находится на одной ЭВМ, а ветвь, которая должна с ним работать – на другой. В отсутствие такого интерфейса пересылку данных необходимо было организовывать вручную, либо полагаться на сетевые возможности операционной системы. По сравнению и с тем, и с другим, MPI-2 гарантирует лучший баланс между универсальностью и быстродействием.
- Сделан шаг в сторону SMP-архитектуры. Теперь разделяемая память может быть не только каналом связи между ветвями, но и местом совместного хранения данных. Для этого ветви делегируют в MPI-2 так называемые буфера-«окна». Интерфейс выполнен так, чтобы в принципе его можно было реализовать и через передачу сообщений в системах, не относящихся к классу SMP. MPI-2 автоматически поддерживает идентичность содержимого всех «окон» с одинаковым идентификатором. Данный механизм назван термином «One-sided communications» («односторонние коммуникации»), так как ветви-получателю не требуется явно вызывать функцию приема для получения новой информации. Передача данных в ветви-отправители осуществляется с применением механизма «Remote memory access» («удаленный доступ к памяти», RMA).
- Рекомендовано использование функций MPI-2 вместо ряда функций стандарта MPI-1.1.
- Расширены многие коллективные процедуры MPI-1.1 для интеркоммуникаторов, предложены дополнительные процедуры для создания интеркоммуникаторов и разработаны две новые коллективные процедуры: обобщенное all-to-all и эксклюзивное сканирование.

Следует отметить что программа, написанная с использованием MPI-1.1, является программой соответствующей MPI-2. В связи с этим исследования новых направлений в развитии программного обеспечения для высокопроизводительных вычислительных систем целесообразно начать с применения проверенной на практике библиотеки коммуникационных функций MPI-1.1.

## 2.7. Выводы

1. Библиотека функций MPI не накладывает ограничений на распределение работ по процессорам.

2. Отладка программ с включенными функциями библиотеки MPI может проводиться в однопроцессорной системе.
3. Минимально необходимый набор функций распараллеливания для любой программы составляют 6 функций библиотеки MPI.
4. Все функции передачи и приема сообщений могут использоваться в любой комбинации друг с другом.
5. Библиотека функций MPI предоставляет множество возможностей для различных реализаций одного и того же вычислительного процесса.

### **3. РАЗРАБОТКА ПРОТОТИПОВ БАЗОВЫХ КОМПОНЕНТОВ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ В БАЗИСЕ ФУНКЦИЙ БИБЛИОТЕКИ МРІ ДЛЯ ОЦЕНКИ СТАТИСТИЧЕСКИХ ХАРАКТЕРИСТИК ВРЕМЕНИ ВЫПОЛНЕНИЯ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ**

Исследование способов организации параллельных вычислительных процессов является актуальной задачей с точки зрения выявления и устранения узких мест в реализации процессов. Для моделирования параллельных программ используются сетевые, иерархические, логические модели, модели с использованием сетей Петри, а так же модели расширения сетей Петри. Для аналитического определения показателей качества функционирования параллельных программ применяются сетевые, иерархические, биологические и логические модели с соответствующими методами их анализа. Для указанного множества моделей разработана группа методов для определения статистических характеристик времени выполнения параллельных программ. Все методы опираются на преобразование графов с одновременным нахождением статистических характеристик. Следующие указанные методы позволяют найти аналитические зависимости времени выполнения параллельных программ при любой топологии параллельной программы:

- 1) Метод преобразования модели параллельной программы с сохранением плотности распределения вероятностей и без сохранения ее структуры;
- 2) Метод преобразования модели параллельной программы с сохранением плотности распределения вероятностей и с сохранением ее структуры;
- 3) Метод свободного объединения графов частных последовательных программ;
- 4) Метод отыскания групп совместных вершин в исходной модели.

Вся система методов позволяет выполнить подтверждение корректности получаемых оценок. Достоинствами приведенной группы методов являются возможности получения информации о зависимости статистических характеристик от состава и содержания выполняемых процедур, от топологии параллельной программы, от механизмов синхронизации функций входов и выходов. Последние три метода, как правило, используются для подтверждения корректности результатов,

полученных с помощью первого указанного метода. Таким образом, возможно формирование вариативного набора параллельных программ и выбора наилучшей программы с точки зрения достижения требуемых характеристик ее времени выполнения, что является основной целью проектирования. Для того, чтобы определить какую функциональную нагрузку будут нести прототипы ключевых компонент программного обеспечения для оценки статистических характеристик времени выполнения параллельных программ необходимо выявить основные процедуры методов. В данной работе рассматривается метод преобразования параллельной программы с сохранением плотности распределения вероятностей и без сохранения ее структуры (метод свертки), который является наиболее гибким и развитым методом.

### **3.1. Типовые процедуры оценки статистических характеристик времени выполнения параллельных программ**

Метод преобразования модели параллельной программы с сохранением плотности распределения вероятностей и без сохранения ее структуры применим для параллельных программ, которые могут быть описаны с помощью сетевых, иерархических, некоторых видов биологических моделей и моделей, использующих темпоральную логику. Уровень методической ошибки может регулироваться и достигать задаваемой величины. Указанный метод позволяет найти полный спектр статистических характеристик времени выполнения параллельной программы (плотность распределения вероятности времени выполнения параллельной программы, функцию распределения вероятности времени выполнения, математическое ожидание, дисперсию и любую другую числовую характеристику). Далее приведены основные процедуры метода для базового варианта сетевой модели.

Определение статистических характеристик времени выполнения состоит из ряда этапов.

- 1) Определение плотности распределения вероятностей времени выполнения последовательных подпрограмм входящих в состав параллельной. Для этого граф частной последовательной программы дополняется псевдовершиной, соответствующей поглощающему состоянию, а затем описывается матрицей переходов:

$$\bar{P}^{(i)} = \begin{vmatrix} P_{1,1}^{(i)} & P_{1,2}^{(i)} & \dots & P_{1,s+1}^{(i)} \\ P_{2,1}^{(i)} & P_{2,2}^{(i)} & \dots & P_{2,s+1}^{(i)} \\ \dots & \dots & \dots & \dots \\ P_{s,1}^{(i)} & P_{s,2}^{(i)} & \dots & P_{s,s+1}^{(i)} \\ 0 & 0 & \dots & 1 \end{vmatrix},$$

где  $(s+1)$  соответствует псевдовершине. Для элементов матрицы справедливы соотношения (3.1)

$$p_{i,j} \in [0,1] \text{ и } \sum_{j=1}^{s+1} p_{i,j} = 1. \quad (3.1)$$

Затем находится плотность распределения вероятности времени выполнения  $i$ -той последовательной программы, входящей в состав параллельной:

$$f_T^{(i)}(k) = P_{1,s+1}^{(i)(k)} - P_{1,s+1}^{(i)(k-1)};$$

$$k = 1, 2, \dots, k_i.$$

$$k_i : \min \left( 1 - \sum_{k=1}^{k_i} (P_{1,s+1}^{(i)(k)} - P_{1,s+1}^{(i)(k-1)}) \right) \leq \delta. \quad (3.2)$$

$$k_i : \min \left( 1 - (\bar{P}^k)_{1,s+1} \right) \leq \delta.$$

Данная операция выполняется для  $i=1,2\dots I$ , где  $I$  – количество последовательных программ, входящих в параллельную программу.

2) Разбиение вырожденной сетевой модели параллельной программы на слои.

Для получения вырожденного графа все графы частных последовательных программ заменяются дугами, в результате чего получается граф, состоящий только из узловых вершин и дуг. Разбиение вырожденного графа на слои осуществляется с соблюдением следующих условий:

- В первый слой включается узловая вершина – точка входа в параллельную программу;
- В последний слой включается узловая вершина, соответствующая точке выхода из параллельной программы;
- Любая узловая вершина текущего слоя не должна иметь предшественников в последующих слоях;
- Узловые вершины внутри каждого слоя не должны иметь связей между собой.

Соблюдение указанных условий приводит к формированию упорядоченной во времени системы процедур объединения и распараллеливания.

3) Нахождение плотности распределения вероятности времени окончания групп объединяемых последовательных параллельно выполняемых подпроцессов. Эта процедура реализуется для каждой узловой вершины. Плотность распределения вероятностей времени окончания группы объединяемых подпроцессов, отображаемых дугами, входящими в узловую вершину последнего слоя, представляет собой плотность распределения вероятности времени выполнения анализируемой параллельной программы. Принимая допущение о том, что время начала выполнения каждой последовательной подпрограммы не зависит от длительности ее выполнения, можно найти плотность распределения вероятности времени окончания каждой отдельной последовательной подпрограммы:

$$f_t^{(i)}(l_i) = \sum_{k_1} f_i^{(i)}(k_1) \cdot f_T^{(i)}(k_2 = l_i - k_1);$$
$$l_i = \min(\tau_i + T_i), \dots, \max(\tau_i + T_i);$$
$$k_1 \rightarrow \tau_i, k_2 \rightarrow T_i,$$
 (3.3)

где  $\tau_i$  – дискретное время начала выполнения  $i$ -той последовательной программы из состава параллельной;

$T_i$  – длительность выполнения  $i$ -той последовательной программы из состава параллельной;

$l_i$  – дискретное время окончания выполнения  $i$ -той последовательной программы из состава параллельной.

Соотношение (3.3) ориентировано на реализацию итеративной процедуры, это значит, что плотность распределения времени начала определенной группы процессов соответствует плотности распределения вероятности времени окончания группы последовательных процессов, отображаемых дугами, входящими в ту узловую вершину, которая реализует распараллеливание.

Для нахождения плотности распределения вероятности времени окончания группы объединенных последовательных процессов необходимо принять допущение о независимости времени окончания объединяемых последовательных программ. Данное предположение не противоречит практике проектирования параллельных программ.

Плотность распределения вероятностей времени окончания группы параллельной программы, в которую входит I последовательных программ описывается соотношением (3.4).

$$\begin{aligned}
 f_j(k) = & f_t^{(1)}(k) \sum_{l_2 \leq k} \dots \sum_{l_I \leq k} f_t^{(2)}(l_2) \cdot f_t^{(3)}(l_3) \cdot \dots \cdot f_t^{(I)}(l_I) + \\
 & + f_t^{(2)}(k) \sum_{l_1 < k} \sum_{l_3 \leq k} \dots \sum_{l_I \leq k} f_t^{(1)}(l_1) \cdot f_t^{(3)}(l_3) \cdot \dots \cdot f_t^{(I)}(l_I) + \dots + \\
 & + f_t^{(I)}(k) \sum_{l_1 < k} \sum_{l_2 < k} \dots \sum_{l_{I-1} < k} f_t^{(1)}(l_1) \cdot f_t^{(2)}(l_2) \cdot \dots \cdot f_t^{(I-1)}(l_{I-1}); \\
 k = & \max_{i} \min(\tau_i + T_i), \dots, \max_{i} \max(\tau_i + T_i); \\
 i = & \overline{1, I}.
 \end{aligned} \tag{3.4}$$

Соотношение (3.4) справедливо при объединении последовательных программ по логической функции «И».

Для логических моделей, когда задействована логическая функция «ИЛИ», необходимо определить процедуру нахождения плотности распределения вероятности времени окончания последовательных программ.

$$\begin{aligned}
 f_j(k) = & f_t^{(1)}(k) \sum_{l_2 \geq k} \dots \sum_{l_I \geq k} f_t^{(2)}(l_2) \cdot f_t^{(3)}(l_3) \cdot \dots \cdot f_t^{(I)}(l_I) + \\
 & + f_t^{(2)}(k) \sum_{l_1 > k} \sum_{l_3 \geq k} \dots \sum_{l_I \geq k} f_t^{(1)}(l_1) \cdot f_t^{(3)}(l_3) \cdot \dots \cdot f_t^{(I)}(l_I) + \dots + \\
 & + f_t^{(I)}(k) \sum_{l_1 > k} \sum_{l_2 > k} \dots \sum_{l_{I-1} > k} f_t^{(1)}(l_1) \cdot f_t^{(2)}(l_2) \cdot \dots \cdot f_t^{(I-1)}(l_{I-1}); \\
 K = & \min_{i} \min(\tau_i + T_i), \dots, \min_{i} \max(\tau_i + T_i); \\
 i = & \overline{1, I}.
 \end{aligned} \tag{3.5}$$

- 4) Исследование статистических характеристик параллельной программы. Для определения статистических характеристик используются соотношения (3.6).

$$\begin{aligned}
 M[T] &= \sum_{k=1}^K k \cdot f_j(k); \\
 K : 1 - \sum_{k=1}^K f_j(k) &\leq \delta; \\
 D[T] &= \sum_{k=1}^K (k - M[T])^2 \cdot f_j(k),
 \end{aligned} \tag{3.6}$$

где  $M[T]$  – математическое ожидание,  $D[T]$  – дисперсия,  $f_j(k)$  – плотность распределения вероятности времени окончания параллельной программы.

Закон распределения можно определить по диаграмме Пирсона на основании значений показателя асимметрии –  $\beta_1$  и показателя островершинности –  $\beta_2$  (3.7).

$$\begin{aligned}
 \beta_1 &= \frac{M_3^2}{M_2^3}, \\
 \beta_2 &= \frac{M_4}{M_2^3},
 \end{aligned} \tag{3.7}$$

где  $M_2$ ,  $M_3$ ,  $M_4$  – соответственно второй, третий и четвертый центральные моменты времени выполнения параллельной программы.

Следует указать причины возможной методической погрешности, присущей рассматриваемому методу:

- Возможное усечение плотности распределения вероятностей длительности выполнения каждой последовательной программы, входящей в состав параллельной;
- Предположение о независимости времени начала каждой последовательной программы и длительности ее выполнения;
- Предположение о независимости времени окончания объединяемых последовательных программ, входящих в состав параллельных;
- Возможное усечение плотности распределения вероятностей времени окончания выполнения объединяемых последовательных программ, входящих в состав параллельных;
- Возможное усечение плотности распределения вероятности времени окончания выполнения параллельной программы.

Обобщая вышеизложенное можно отметить, что для моделирования параллельных процессов с произвольными функциями объединения последовательных фрагментов применяются многоуровневые логические модели. Как правило, в большинстве реальных систем с параллельной обработкой информации и принятия решений для комплексирования

последовательных подпроцессов используются такие логические функции как «И», «ИЛИ» или более общий случай – совмещенное использование «И-ИЛИ».

### 3.2. Обоснование выбора технических средств

Рассмотрим основные бесплатные реализации библиотеки MPI:

- 1) MPICH-переносимая реализация (работает почти на всех UNIX-системах и Windows NT), разработанная в Argonne National Laboratory. Поддерживаются кластеры на базе SMP-узлов. Поддерживается стандарт MPI 1.1 и некоторые элементы стандарта MPI 2.0.
- 2) WMPI-реализация MPI для платформ Win32 (Microsoft Windows 95/98/NT), разработанная и поддерживаемая Jose Meireles Marinho (Университет Coimbra, Португалия). Базируется на реализации P4 для Win32 (интерфейс к P4 также входит в поставку). Последняя версия – WMPI 1.3; поддерживается стандарт MPI 1.1, включена библиотека ROMIO от ANL, реализующая спецификацию MPI I/O стандарта MPI 2.0. Реализация совместима с MPICH 1.1.2 (т.е. возможна организация гетерогенных кластеров UNIX/Win32). Поставляются только двоичные файлы для Win32 (Intel и Alpha).
- 3) MP-MPICH-мультиплатформенная реализация MPI на базе MPICH. Включает NT-MPICH (версию MPICH для Windows NT) и SCI-MPICH (версию MPICH для SCI-коммутаторов). Разработка RWTH-Aachen (Аахен, Германия).

Для сравнения указанных реализаций на рис.3.1 и рис.3.2 приведены графики, показывающие эффективность работы реализаций через технологию Fast Ethernet на тестах межпроцессорных обменов Pallas MPI Benchmark [36]. Сравнивались пакеты NT-MPICH 1.2, MPICH.NT 1.2.1, MPI/PRO 1.6 (комерческая реализация MPI) и WMPI 1.53. Тестирование проводилось на машинах с процессором Pentium 2, 450 MHz. Полные характеристики тестов приводятся на сайте <http://www.lfbs.rwth-aachen.de>. В настоящее время в открытом доступе есть аналогичные тесты, разработанные в НИВЦ МГУ [37]. Указанный пакет включает четыре теста, три из которых тестируют эффективность среды (сети) передачи данных между процессорами (узлами), а четвертый тестирует производительность совместного доступа узлов к сетевому файл-серверу. В отечественный пакет входят: transfer – тест латентности и скорости пересылок между двумя узлами; nettest – тест пропускной способности сети при сложных обменах по различным логическим топологиям; mpitest

- тест эффективности основных операций MPI; nfstest – тест производительности файл-сервера. По данным тестов Pallas MPI Benchmark наиболее производительным является реализация MPICH.NT 1.2.1.

Разработка прототипов базовых компонентов программного обеспечения в базисе функций библиотеки MPI для оценки статистических характеристик времени выполнения параллельных программ проводилась в среде операционной системы семейства Windows NT, как наиболее распространенной на данном этапе развития вычислительных систем. На данный момент доступна версия MPICH.NT 1.2.5, поэтому разработка прототипов базовых компонентов осуществлялась с его использованием. После запуска файла инсталляции «mpich.nt.1.2.5.exe» следует стандартная процедура инсталляции приложения Windows. Для разработки ключевых компонент установлены: утилиты пакета (основные – MPD, MPIConfig, MPIRun, guiMPIRun); библиотека MPI SDK для Windows; библиотека MPI SDK для Linux. Неполная документация по SDK, документация по утилитам и документация по созданию проекта с использованием библиотеки в среде MS Developer Studio – Visual C++.

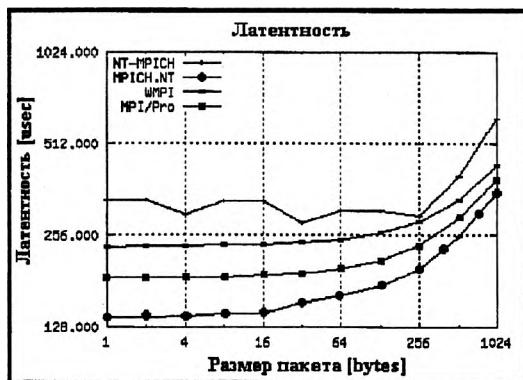


Рис. 3.1. Зависимость латентности от размера пакетов по результатам теста Pallas MPI Benchmark

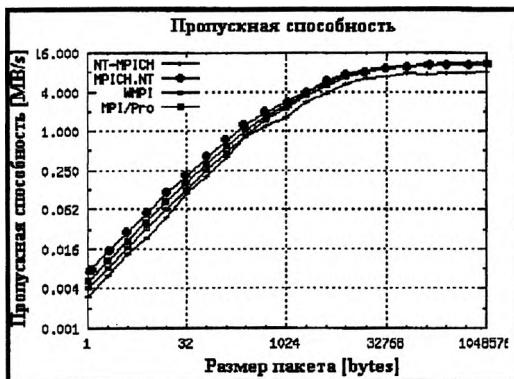


Рис. 3.2. Зависимость пропускной способности от размера пакетов по результатам теста Pallas MPI Benchmark

### 3.3. Программный прототип объединения подпрограмм по логической функции «И»

#### 3.3.1. Функциональная спецификация прототипа

Работу программного прототипа можно разделить на три этапа и составить функциональную спецификацию для каждого из них.

Первый последовательный фрагмент параллельной программы должен производить вычислительные операции для нахождения  $f_t^{(i)}(k)$ ;  $k = 1, 2, \dots, K^{(i)}$  плотности распределения вероятностей дискретной длительности выполнения последовательной программы, смена состояний которой описывается стохастической матрицей вероятностей переходов  $\bar{P}^{(i)}$  размерности  $(S_i + 1) \times (S_i + 1)$  в дискретном пространстве состояний. На данном этапе должна быть предусмотрена возможность усечения распределения согласно задаваемой погрешности  $\delta_i$  и передача найденных значений плотности распределения вероятности дискретной длительности  $2, \dots, (n-1)$  параллельно выполняемым последовательным фрагментам.

В функциональную спецификацию 2-го, 3-го и  $(N-1)$ -го параллельно выполняемых последовательных фрагментов программы должны быть включены вычислительные операции для нахождения  $f_t^{(i)}(k)$ ;  $k = 1, 2, \dots, K^{(i)}$ ;  $i = 2, \dots, (N-1)$  плотности распределения дискретной длительности выполнения соответствующей последовательной программы, смена состояний которой описывается стохастической

матрицей вероятностей переходов  $\bar{P}^{(i)}$  размерности  $(S_i + 1) \times (S_i + 1)$ ;  $i = 2, \dots, (N - 1)$  в дискретном пространстве состояний. Так же в каждом из параллельно выполняемых фрагментов должна быть предусмотрена возможность усечения распределения согласно задаваемой погрешности  $\delta_i$ ;  $i = 2, \dots, (N - 1)$ . Все параллельно выполняемые фрагменты реализуют вычислительные операции для нахождения  $f_i^{(i)}(l_i)$ ;  $i = 2, \dots, (N - 1)$  плотности распределения дискретного времени окончания выполнения соответствующей последовательной программы при условии ее начала в момент времени завершения первого последовательного фрагмента, характеризующийся плотностью распределения,  $f_i^{(1)}(l_i) = f_i^{(1)}(k)$ ;  $k = 1, \dots, K^{(1)}$ . Все найденные значения плотностей распределения дискретных времен окончания передаются последнему  $N$ -ому последовательному фрагменту разрабатываемого прототипа.

Последний  $N$ -ый последовательный фрагмент параллельной программы вычисляет  $f_t^{(N)}(k)$ ;  $k = 0, 1, 2, \dots, K^{(N)}$  плотности распределения вероятностей дискретной длительности выполнения последовательной программы, смена состояний которой описывается стохастической матрицей вероятностей переходов  $\bar{P}^{(N)}$  размерности  $(S_N + 1) \times (S_N + 1)$  в дискретном пространстве состояний, с возможностью усечения распределения согласно задаваемой погрешности  $\delta_N$ . В этом же фрагменте реализуются вычислительные операции для нахождения  $f_2(k); k = 1, 2, \dots, K^{(2)}$  плотности распределения вероятностей дискретного времени окончания выполнения параллельной программы, при условии начала  $N$ -ого фрагмента программы в момент времени завершения всех параллельно выполняемых фрагментов.

Последний фрагмент реализует вычисление статистических характеристик времени выполнения параллельной программы, обеспечивающей объединение последовательных подпрограмм согласно функции «И», по соотношениям (3.6).

С учетом всех соотношений, приведенных в подразделе 3.1, вычисление объединения распределений для  $N$  параллельных процессов осуществляется по формуле (3.8)

$$f^{AND}(k) = \prod_{i=1}^N \sum_{j=0}^k f^{(i)}(j) - \prod_{i=1}^N \sum_{j=0}^{k-1} f^{(i)}(j). \quad (3.8)$$

Демонстрацию решения поставленной задачи будем производить на частном случае описания последовательных фрагментов параллельной программы

$$\bar{P}^{(2)} = \dots = \bar{P}^{(N-1)} = \begin{bmatrix} 0 & (1-P_1) & 0 & \dots & 0 & \dots & 0 & P_1 \\ 0 & 0 & (1-P_2) & \dots & 0 & \dots & 0 & P_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & (1-P_j) & \dots & 0 & P_j \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\ (1-P_M) & 0 & 0 & \dots & 0 & \dots & 0 & P_M \\ 0 & 0 & 0 & \dots & 0 & \dots & 0 & 1 \end{bmatrix}, \quad (3.9)$$

$$S_2 = \dots = S_{N-1} = M.$$

При этом аналитические выражения для определения плотностей распределения вероятностей

$$f_T^{(n)}(k); \quad k=1,2,\dots,K^{(n)}$$

$f_2(k); \quad k=\max_n(\min_2, \min_3, \dots, \min_{(N-1)}), \dots, \max_n(\max_2, \max_3, \dots, \max_{(N-1)}) \quad n=2,\dots,(N-1)$  находятся при условии, что

$$f_T^{(1)}(k) = \begin{cases} 1, & k=0 \\ 0, & k \in [1, \infty) \end{cases}$$

В этом частном случае, когда  $\bar{P}^{(2)} = \bar{P}^{(3)} = \dots = \bar{P}^{(N-1)}$ ,  $S_2 = \dots = S_{N-1}$ , используется обозначение  $P$  для матрицы. Соотношение (3.8) модифицируется исходя из этих условий. Рассмотрим сумму (с учетом того, что матрицы одинаковы)

$$\sum_{j=0}^k f^{(i)}(j) = (\bar{P}^0)_{1,n+1} + (\bar{P}^1)_{1,n+1} - (\bar{P}^0)_{1,n+1} + \dots + (\bar{P}^k)_{1,n+1} - (\bar{P}^{k-1})_{1,n+1} = (\bar{P}^k)_{1,n+1},$$

тогда (3.8) запишется как:

$$f^{AND}(k) = \prod_{i=1}^N (\bar{P}^k)_{1,n+1} - \prod_{i=1}^N (\bar{P}^{k-1})_{1,n+1},$$

следовательно:

$$f^{AND}(k) = ((\bar{P}^k)_{1,n+1})^N - ((\bar{P}^{k-1})_{1,n+1})^N.$$

### **3.3.2. Альтернативные варианты реализации программного прототипа**

Для выбора структуры и алгоритма работы прототипа далее рассмотрены некоторые возможные реализации модели прототипа.

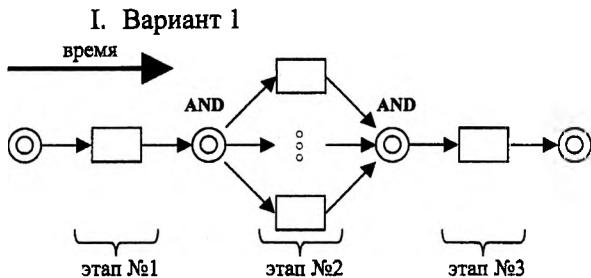


Рис. 3.2. Ориентированный граф сетевой модели параллельной программы для варианта 1

## Алгоритм работы.

## Этап №1.

1. Главный процесс считывает из файла требуемую точность вычислений и запрашивает у пользователя имя файла содержащего матрицы, либо запрашивает завершение программы.
  2. Главный процесс считывает из файла матрицы и проверяет их корректность, в случае некорректных данных программа завершается.
  3. Главный процесс передает точность и матрицы всем остальным процессам (коммуникация осуществляется функциями MPI\_Send и MPI\_Recv).
  4. Главный процесс вычисляет  $f_T^{(1)}(k)$ (с учетом заданной точности) и передает полученный вектор всем процессам (коммуникация осуществляется функциями MPI\_Bcast).

## Этап №2.

5. Все процессы (включая главный) вычисляют  $f_t^{(i)}(k)$  и свертки  $f_t^{(i)}(l_i)$ , полученные результаты передаются главному процессу (коммуникация осуществляется функциями MPI\_Send и MPI\_Recv).

### Этап №3.

6. Главный процесс вычисляет свертку по AND полученных результатов.

- Главный процесс вычисляет  $f_T^{(N)}(k)$  и свертку результата согласно функции AND.
- Главный процесс вычисляет математическое ожидание, дисперсию и реализует вывод результата.

## II. Вариант 2

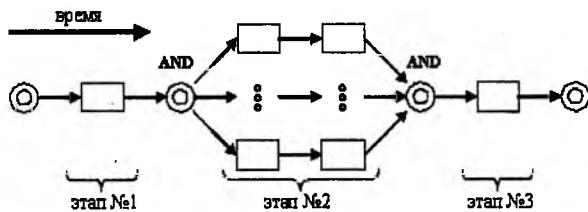


Рис. 3.3. Ориентированный граф сетевой модели параллельной программы для варианта 2

Алгоритм работы.

Этап №1.

- Главный процесс считывает из файла требуемую точность вычислений и запрашивает у пользователя имя файла, содержащего матрицы, либо запрашивает завершение программы.
- Главный процесс считывает из файла матрицы и проверяет их корректность, в случае некорректных данных программа завершается.
- Главный процесс передает точность и матрицы всем остальным процессам (коммуникация осуществляется функциями MPI\_Send и MPI\_Recv).
- Главный процесс вычисляет  $f_T^{(1)}(k)$  (с учетом заданной точности) и передает полученный вектор всем процессам (коммуникация осуществляется функциями MPI\_Bcast).

Этап №2.

- Все процессы (включая главный) вычисляют по две  $f_T^{(l)}(k)$  и две свертки  $f_T^{(l)}(l_i)$ , полученные результаты передаются главному процессу (коммуникация осуществляется функциями MPI\_Send и MPI\_Recv). Если исходное число процессов нечетное, то главный процесс будет обрабатывать на этапе №2 только одну матрицу.

### Этап №3.

6. Главный процесс вычисляет свертку по AND полученных результатов.
7. Главный процесс вычисляет  $f_r^{(N)}(k)$  и свертку результата согласно функции AND.
8. Главный процесс вычисляет математическое ожидание, дисперсию и осуществляет вывод результата.

### III. Вариант 3

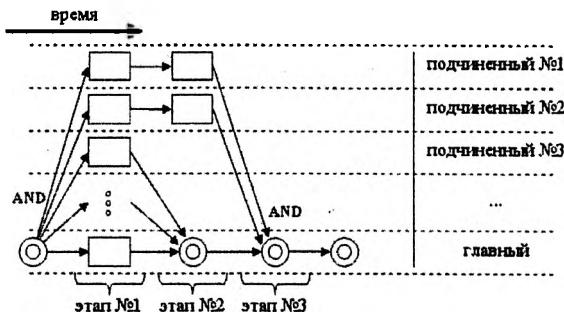


Рис. 3.4. Ориентированный граф сетевой модели параллельной программы прототипа для варианта 3

Алгоритм работы.

### Этап №1.

1. Главный процесс считывает из файла требуемую точность вычислений и запрашивает у пользователя имя файла, содержащего матрицы, либо запрашивает завершение программы.
2. Главный процесс считывает из файла матрицы и проверяет их корректность, в случае некорректных данных программа завершается.
3. Главный процесс передает точность и матрицы всем остальным процессам (коммуникация осуществляется функциями MPI\_Send и MPI\_Recv).

### Этап №2.

1. Главный процесс вычисляет первую матрицу, параллельно все остальные процессы вычисляют вторую.
2. Главный процесс получает первую матрицу от всех.

### Этап №3.

1. Главный процесс вычисляет все остальные вычисления.
2. Главный процесс вычисляет математическое ожидание, дисперсию и осуществляет вывод результата.
3. Программа возвращается к шагу 1.

#### 3.3.3. Сравнение приведенных топологий

Во втором варианте уменьшится (при одинаковом числе матриц) число процессов в 2 раза по сравнению с первым вариантом. Более корректный результат сравнения вариантов 1 и 2 следует ожидать при четном числе процессов. Можно предположить, что второй вариант реализации будет работать быстрее в случае, когда временные затраты на вычисления меньше коммуникационных затрат. Более того, чем меньше эти затраты, тем больше вычислений следует производить на одном процессоре, модифицируя программу аналогично преобразованию варианта №1 в вариант №2 (в пределе получим последовательную программу).

Поскольку в случае малых временных затрат на вычисления схема действий ясна (уменьшение параллельности – варианты 1 и 2), вариант 3 представляет противоположный случай. Если временные затраты на вычисления гораздо больше коммуникационных затрат и число процессов фиксировано, необходимо устранять простой процессов.

Первый простой происходит при ожидании вычисления главным процессом первого распределения, поэтому топология изменена следующим образом: реализована пересылка матриц в начале работы программы и запуск вычисления параллельно (первое распределение вычисляется одновременно со вторым во всех подчиненных процессах), затем организована пересылка полученных данных в основной процесс, все остальные вычисления реализованы в основном процессе.

Второй простой (менее значительный, чем первый) происходит из-за реализации приема главным процессом данных от подчиненных процессов: прием происходит в фиксированном порядке – сначала принимаются данные первого процесса. Поэтому прием модифицирован таким образом, чтобы первыми данные принимались от процесса, закончившего вычисления первым, затем от процесса, закончившего вычисления вторым, и т.д.

Если предположить, что размерности всех матриц – величины одного порядка, то более рациональной представляется топология, реализующая логику варианта 3.

Следует отметить, что рассуждения, проведенные при создании различных вариантов, справедливы для данной конкретной конфигурации системы «сеть+процессоры». В двух разных вариантах конфигурации при одинаковых исходных данных (матрицах) временные затраты на коммуникации могут быть больше затрат на вычисления в одной системе и меньше в другой. Что неизбежно приведет к разным результатам в оценке эффективности того или иного варианта работы программы.

### 3.3.4. Построение схемы измерения временных характеристик

Для сравнения различных вариантов реализуемого прототипа необходимо оценить временные характеристики выполнения прототипов на одинаковых наборах исходных данных. В качестве функции для измерения временных характеристик используется функция

double MPI\_Wtime(void), которая возвращает время в секундах, прошедшее с некоторого момента в прошлом в виде числа типа double. Так же воспользуемся функцией double MPI\_Wtick(void) для определения точности (разрешения) функции MPI\_Wtime().

Для измерения временных затрат используется разница между двумя значениями, полученными в результате вызова функции MPI\_Wtime() до и после некоторого события.

Пример кода, иллюстрирующего подобную процедуру:

```
// Логический блок
{
    double rTimeBefore; // время до события
    double rTimeDelta; // временные затраты на событие
    // измеряем текущее время
    rTimeBefore = MPI_Wtime();
    // собственно событие
    ...
    // вычисляем разницу времени - искомые затраты
    rTimeDelta = MPI_Wtime() - rTimeBefore;
}
```

При последовательных измерениях временных затрат возможен другой вариант с целью уменьшения числа дублирующихся вызовов. Сначала в ключевых точках измеряется время, данные сохраняются в массиве, который подлежит последующей обработке.

Пример кода, иллюстрирующего подобную процедуру:

```
// Логический блок
{
    double raTime[nNumPoints]; // временные метки
    // измеряем текущее время
    raTime[0] = MPI_Wtime();
    // собственно событие №1
    ...
    // измеряем текущее время
    raTime[1] = MPI_Wtime();
    ...
    // измеряем текущее время
    raTime[nNumPoints-2] = MPI_Wtime();
    // собственно событие №«nNumPoints-1»
    ...
    // измеряем текущее время
    raTime[nNumPoints-1] = MPI_Wtime();
    // обработка результатов
    ...
}
```

Временные затраты на вызов функции MPI\_Wtime() и вычисление разницы двух времен можно считать пренебрежимо малыми. Обработка данных проводится в основном процессе.

В качестве меток или точек измерения, позволяющих измерить вычислительные и коммуникационные временные затраты для всех элементов, используются все «узловые» точки программы, т.е. выполняются измерения времени во всех промежутках между вычислениями и коммуникациями, а также в начале и в конце программы.

Следует отметить, что при таком выборе гарантируется точное измерение времени, затраченного на вычисления. Но в полученном значении времени, затраченном на коммуникации, также учитывается время «простоя» процессов – время, затраченное на ожидание начала приема или передачу сообщений, которое неизбежно при всех указанных выше топологиях.

### 3.3.5. Методика тестирования, проведение эксперимента и представление результатов

Для полноценного тестирования требуется создание достаточно больших размерностей стохастических матриц вероятностей переходов (например, размерностью 100x100 и более). Для этого разработана и реализована на языке С программа генерации матриц произвольной (до 10000x10000) размерности. Генерируемые программой стохастические матрицы основаны на базовой структуре стохастических матриц

вероятностей переходов вида (3.9), где значения  $p_i$ ,  $i = 1 \dots s$ , выбираются случайными из интервала (0, 1).

Для упрощения работы с программой она дополнена кодом, запускающим основной вычислительный блок несколько раз для каждого размера матриц из группы матриц, начиная с матрицы размером 5x5.

Ниже приведена часть кода, выполняющая автоматическое тестирование:

```
char    *cFN[]=
{"5_","10_","15_","20_","25_","30_","35_","40_","45_","50_","55_",
", "60_","65_","70_","75_","80_","85_","90_","95_","100_","105_",
"110_","115_","120_","125_","130_","135_","140_","150_","160_",
"170_","180_","190_","200_"},;

#ifndef nsAUTOTEST
for (int nI = 0; nI < sizeof(cFN)/sizeof(*cFN); nI++)
{
    NsCallMain(nNumPrc, nCurrID, cFN[nI]);
    NsCallMain(nNumPrc, nCurrID, cFN[nI]);
    NsCallMain(nNumPrc, nCurrID, cFN[nI]);
    NsCallMain(nNumPrc, nCurrID, cFN[nI]);
}
#else
while (!bDone)
{
    bDone = NsCallMain(nNumPrc, nCurrID, NULL);
}
#endif
```

В случае, если определена переменная препроцессора «nsAUTOTEST», данный код выполняет основной блок 4 раза для каждого элемента массива «cFN», который содержит имена файлов, содержащих матрицы (имя файла совпадает с размерностью матрицы, которую он содержит).

Таким образом, для каждого варианта реализации программы и числа процессов, участвующих в данном запуске, пользователю достаточно просто запустить вычисления.

В результате работы программы выдаются данные, показанные на рис.3.5 (приведено начало работы 1-го варианта на одном компьютере):

```

Number of processes: 5 Run Advanced Options
Output:
Process 0, started. My ID = 1
Process 1, started. My ID = 2
Process 2, started. My ID = 3
Process 3, started. My ID = 4
Process 4, started. My ID = 5
:ReconfigureInitialPrecision=precision=0.0000010000000000
Среднее = 17.2324, дисперсия = 32.1089
Процесс 0, время = 0.00010512, вычисления = 5
(10487.942141, 10487.942203, 10487.944072, 10487.944072, 10487.940726,
Размерность суммарной: 0.00010512, вычисления суммарные: 0.0004374
Процесс 0, время = 0.04682165, вычисления =
(10487.939564, 10487.940313, 10487.9404623, 10487.9410009, 10487.9487863,
Коммуникации суммарны: 0.00010512, вычисления суммарные: 0.0027143
Процесс 4, время = 0.00159812, вычисления =
(10487.9421728, 10487.9421761, 10487.9423472, 10487.9454789, 10487.9437660,
Размерность суммарной: 0.00159812, вычисления суммарные: 0.00053608
Процесс 3, время = 0.03893875, вычисления =
(10487.9427410, 10487.9427452, 10487.9809208, 10487.9814577, 10487.9816776,
Коммуникации суммарны: 0.03893875, вычисления суммарные: 0.00054337
Среднее = 17.2324, дисперсия = 32.1089
#444444 Вычисления завершены #444444 (Cells = 5)
Процесс 0, время = 0.00010512, вычисления =
(10487.942141, 10487.942203, 10487.944072, 10487.944072, 10487.940726,
Коммуникации суммарны: 0.00010512, вычисления суммарные: 0.0004374

```

Рис. 3.5. Пример выходных данных прототипа

В процессе исследования разработана и реализована на языке «Perl» утилита, производящая чтение выходных данных такого типа в память и создания сводной таблицы.

При обработке выходных данных, содержащих исходные времена, производится усреднение величин, полученных при разных итерациях на одинаковых начальных условиях.

Пример файла, генерируемого утилитой (3-й вариант программы, 3 компьютера, часть файла):

```

m: 5, p:0 0.002377 0.000769 0.001608 p:1 0.000849 0.000582
0.000267 p:2 0.000802 0.000401 0.000401
m: 10, p:0 0.003666 0.001258 0.002407 p:1 0.001740 0.001204
0.000536 p:2 0.001965 0.000802 0.001163

```

Формат вывода данной утилиты следующий:

число после «m:» показывает размерность матрицы, после «p:» написан номер процесса, далее – три временных показателя для него (все время, время, затраченное на коммуникации и время, затраченное на вычисления).

Основная цель разработки данной утилиты заключается в необходимости создания различных сводных таблиц по полученным данным (без модификации исходного кода основной программы и повторного запуска для получения статистики другого вида).

Полный исходный текст утилиты для преобразования выходной информации и текст генератора матриц приведены в Приложении 2.

Для тестирования созданных вариантов прототипов необходимо выполнить следующее:

1) Общее тестирование результатов работы прототипов:

- тестирование результатов работы прототипов на одной исходной матрице;
- тестирование результатов работы прототипов на системе матриц.

2) Тестирование качества работы прототипов.

1) Общее тестирование результатов работы прототипов:

a) Исходная матрица  $P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$ .

Для такой матрицы итоговая величина математического ожидания равна 6, а дисперсия равна 0 для любого числа процессов. Следует отметить, что результат не зависит и от точности, поскольку элементы матрицы, являются целыми числами.

Результат работы прототипа:

```
.....  
Введите имя файла содержащего матрицу (*.mtr), or 'q' for quit  
6  
.....  
Среднее = 6.0000, дисперсия = 0.0000  
##### Вычисления завершены #####  
(cols = 3)
```

b) Исходная матрица  $P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$ .

Для такой матрицы итоговая величина математического ожидания равна 3, а дисперсия равна 0 для любого числа процессов.

Результат работы прототипа:

```
.....  
Введите имя файла содержащего матрицу (*.mtr), or  
'q' for quit  
6  
Среднее = 3.0000, дисперсия = 0.0000  
##### Вычисления завершены ##### (cols = 3)
```

### c) Тест системы матриц.

Исходные матрицы:

$$\begin{pmatrix} 0 & 0.2 & 0 & 0 & 0.8 \\ 0 & 0 & 0.2 & 0 & 0.8 \\ 0 & 0 & 0 & 0.2 & 0.8 \\ 0.2 & 0 & 0 & 0 & 0.8 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0.7 & 0 & 0.3 \\ 0 & 0 & 0.7 & 0.3 \\ 0.5 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0.2 & 0 & 0 & 0.8 \\ 0 & 0 & 0.3 & 0 & 0.7 \\ 0 & 0 & 0 & 0.2 & 0.8 \\ 0.3 & 0 & 0 & 0 & 0.7 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0.3 & 0 & 0 & 0.7 \\ 0 & 0 & 0.2 & 0 & 0.8 \\ 0 & 0 & 0 & 0.3 & 0.8 \\ 0.2 & 0 & 0 & 0 & 0.8 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0.5 & 0.5 \\ 0.6 & 0 & 0.4 \\ 0 & 0 & 1 \end{pmatrix}$$

Заданная точность для вычисления:  $\delta = 0.0000001$ .

Результат работы прототипа:

```
Process on termit., started. My ID = 2
Process on termit., started. My ID = 0
::NsConfigInitialize::precision 0.0000010000000000
.....  
процесс 0, дельта: 0.00734088, тайминг:  
{13044.0378445, 13044.0378900, 13044.0382158,  
13044.0383700, 13044.0445205, 13044.0451854};  
коммуникации суммарно: 0.00647624, вычисления суммарно:  
0.00086464  
.....  
Process on termit., stopped
Process on termit., stopped
Процесс 2, дельта: 0.00215782, тайминг:  
{13044.0385968, 13044.0386004, 13044.0404149,  
13044.0405775, 13044.0407521, 13044.0407546};  
коммуникации суммарно: 0.00198908, вычисления суммарно:  
0.00016874
Process on termit., stopped
Среднее = 6.6313, Дисперсия = 6.9017
##### Вычисления завершены #####
```

### 2) Тестирование качества работы прототипов.

При тестировании качества работы прототипов исследуется общее время решения поставленной задачи, промежуточные результаты (временные затраты на каждый элемент топологии) используются для анализа.

Фактически реализован запуск задачи одной сложности (например, задачу, состоящую из системы 6 матриц) на трех вариантах программы.

При этом число используемых процессоров в рассматриваемых вариантах реализации разное: например для шести матриц первый и третий вариант требуют 4 процессора, второй – 2 процессора.

Для минимизации случайной флюктуации результата проводится несколько экспериментов в каждом варианте реализации для любой задачи. Конечным результатом считается среднее значение.

Для упрощения анализа результатов на рис.3.6, рис. 3.7, рис. 3.8 приводятся графики, построенные по обработанным данным, полученным при запусках прототипов.

Из приведенных графиков видно, что третий вариант действительно является самым быстрым, то есть предположения относительно «узких» мест исходной программы оказались верны.

Результаты тестирования, показывающие затраты на коммуникации, приведены на рис.3.9, 3.10, 3.11, значения времени вычисления показаны на рис. 3.12, 3.13, 3.14.

На рис. 3.9 видно, что коммуникации в третьем и втором вариантах занимают больше времени. Меньше всего времени на обмены требуется варианту 1.

При сопоставлении времен, затраченных на коммуникации и вычисления, выясняется, что баланс между ними наблюдается у варианта №3.

Из графиков, показывающих временные затраты на вычисления (рис.3.15, 3.16, 3.17), видно, что исключение вычисления двух матриц из главного процесса сильно уменьшает затраченное время. Именно это и является причиной быстрой работы 3-го варианта.

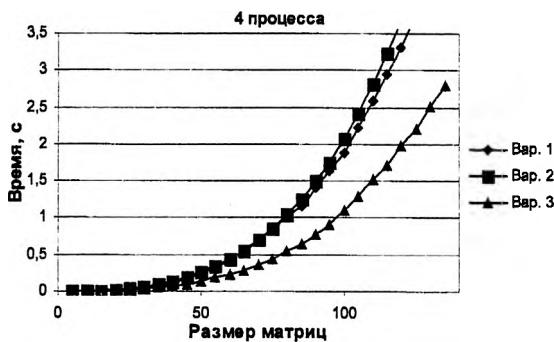


Рис. 3.6. Зависимости общего времени выполнения четырех процессов от размера матриц

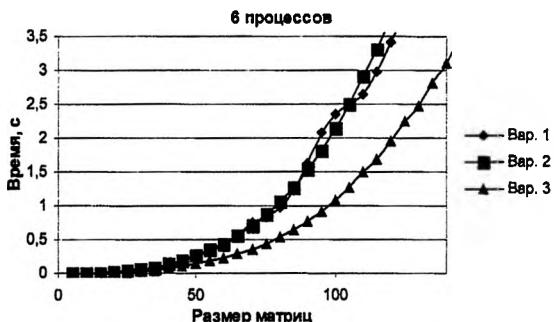


Рис. 3.7. Зависимости общего времени выполнения шести процессов от размера матриц

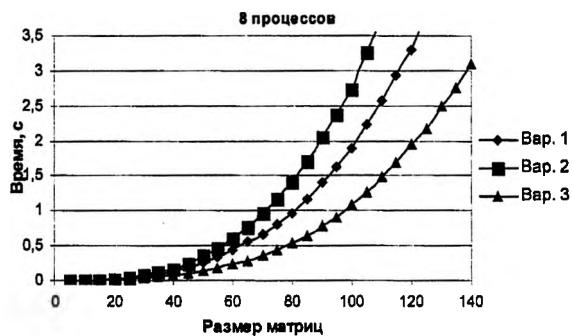


Рис. 3.8. Зависимости общего времени выполнения восьми процессов от размера матриц

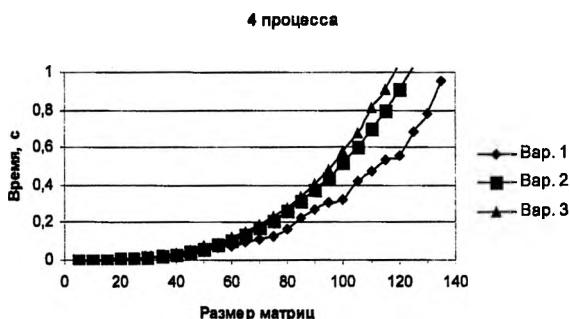


Рис. 3.9. Зависимости времени на коммуникационные обмены четырех процессов от размера матриц

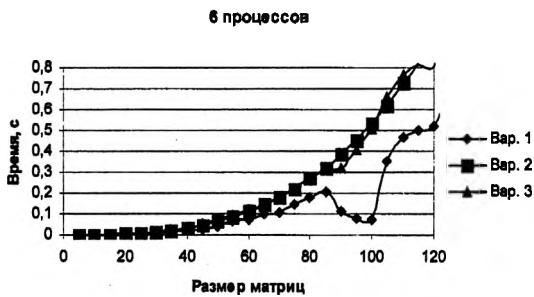


Рис. 3.10. Зависимости времени на коммуникационные обмены шести процессов от размера матриц



Рис. 3.11. Зависимости времени на коммуникационные обмены восьми процессов от размера матриц

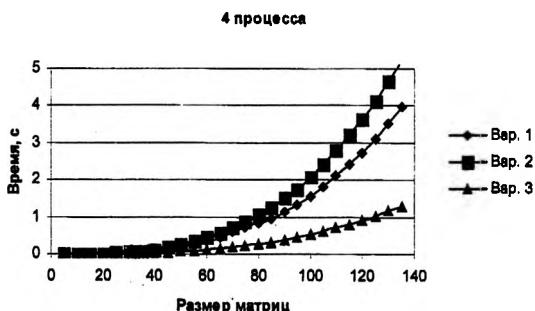


Рис. 3.12. Зависимости времени вычисления для четырех процессов от размера матриц

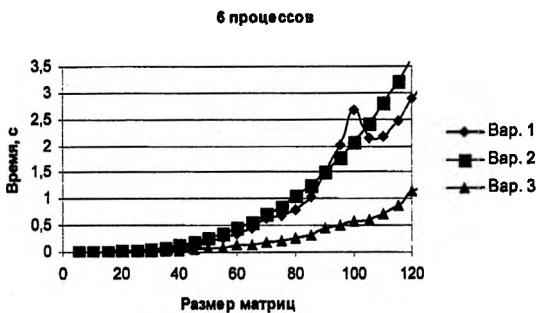


Рис. 3.13. Зависимости времени вычисления для шести процессов от размера матриц

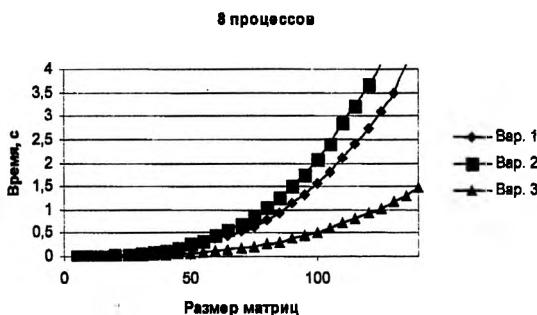


Рис. 3.14. Зависимости времени вычисления для восьми процессов от размера матриц

Для того, чтобы определить самый быстрый вариант при матрицах размера 5x5 (и меньше), на рис. 3.15 представлены три увеличенных фрагмента графиков, приведенных на рис. 3.6, 3.7, 3.8.

На данном участке при 4-х и 6-ти процессах зависимость практически одна и та же, но при 8 процессах вариант №2 быстрее №1. Наименьшее время показывает вариант №3. Это объясняется тем, что при очень маленьких матрицах очень много времени (по сравнению с вычислениями распределений) занимает вычисление сверток.

Из выше сказанного можно сделать вывод, что из трех реализованных вариантов лучшими временными характеристиками обладает третий.

На рис.3.16, 3.17, 3.18 показаны сравнительные зависимости времен от размера матриц при работе на различных реализациях локальной сети.

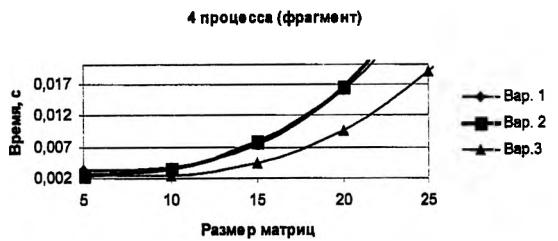


Рис. 3.15. Фрагменты общего времени выполнения альтернативных вариантов программного прототипа от размера матриц

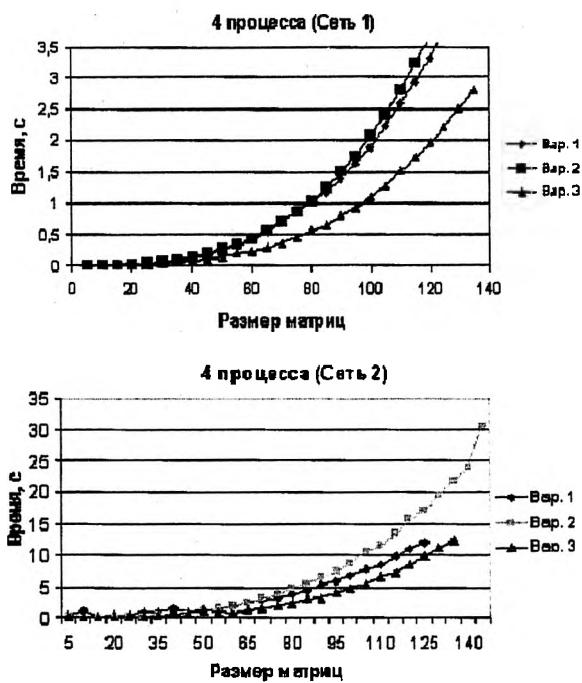


Рис. 3.16. Зависимость общего времени выполнения вариантов программного прототипа от размера матриц на разных сетях

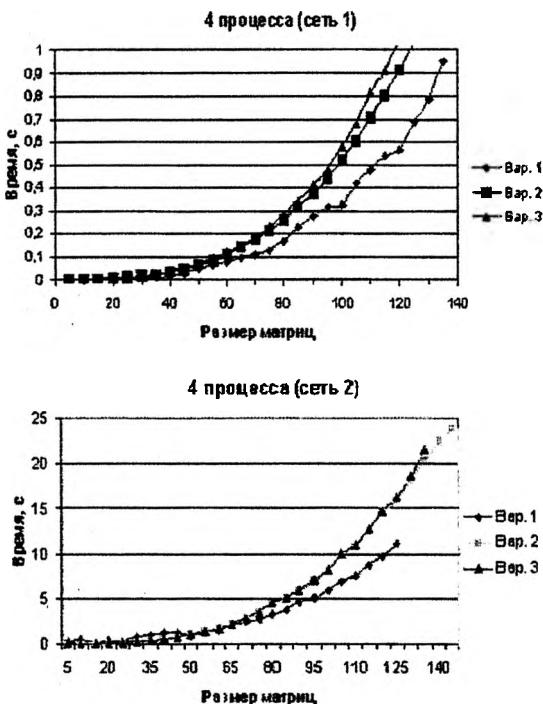


Рис. 3.17. Зависимость времени коммуникационных обменов от размера матриц на разных сетях

На указанных рисунках не следует сравнивать временные показатели в числовом виде, так как во второй сети использовалось более медленное оборудование, стоит сравнивать только общие тенденции.

При сравнении общего времени выполнения прототипов на обеих сетях получаются схожие результаты (см. рис. 3.16).

При сравнении времен коммуникационных обменов ситуация так же схожа (см. рис.3.17). Однако при сравнении времени вычисления вариант №2 на второй сети выполняется быстрее (см. рис.3.18).

На всех приведенных диаграммах третий вариант реализации так же показывает лучший результат.

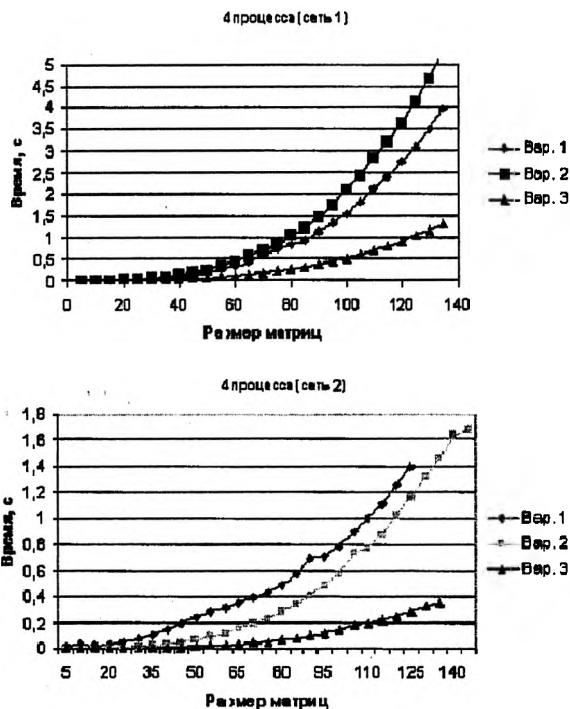


Рис. 3.18. Зависимость времени вычисления от размера матриц на разных сетях

### 3.4. Программный прототип объединения подпрограмм по логической функции «ИЛИ»

В данном разделе рассматриваются две реализации прототипа. Первый предполагает вычислительные операции для нахождения сверток вычислительных процессов и статистических характеристик, второй предназначен только для вычисления статистических характеристик.

#### 3.4.1. Функциональная спецификация прототипов

##### Вариант 1.

Функциональная спецификация прототипа 1 аналогична приведенной в подразделе 3.3.1. Отличается только последний этап вычислений. В этом фрагменте реализуются вычислительные операции для нахождения  $f_2(k); k = 1, 2, \dots, K^{(P)}$  плотности распределения

вероятностей дискретного времени окончания выполнения параллельной программы, при условии начала N-ого фрагмента программы в момент времени завершения всех параллельно выполняемых фрагментов.

Последний фрагмент реализует вычисление статистических характеристик времени выполнения параллельной программы, обеспечивающей объединение последовательных программ согласно функции «ИЛИ», по соотношениям (3.6).

Вычисление объединения распределений по логической функции «ИЛИ» для N параллельных процессов осуществляется по формуле (3.10)

$$f^{OR}(k) = \prod_{l=1}^N \left( 1 - \sum_{j=0}^{k-1} f^{(l)}(j) \right) - \prod_{l=1}^N \left( 1 - \sum_{j=0}^k f^{(l)}(j) \right) . \quad (3.10)$$

Аналитическое выражение для нахождения объединения по «ИЛИ» в частном случае, когда  $\bar{P}^{(2)} = \bar{P}^{(3)} = \dots = \bar{P}^{(N-1)}$ ,  $S_2 = \dots = S_N$  выводится из формулы (3.10). Далее используется обозначение  $P$  для матрицы. Рассмотрим сумму с учетом того, что матрицы одинаковы

$$\sum_{j=0}^k f^{(l)}(j) = (\bar{P}^0)_{1,s+1} + (\bar{P}^1)_{1,s+1} - (\bar{P}^0)_{1,s+1} + \dots + (\bar{P}^k)_{1,s+1} - (\bar{P}^{k-1})_{1,s+1} = (\bar{P}^k)_{1,s+1},$$

тогда (3.10) записывается как:

$$f^{OR}(k) = \prod_{l=1}^N \left( 1 - (\bar{P}^{k-1})_{1,s+1} \right) - \prod_{l=1}^N \left( 1 - (\bar{P}^k)_{1,s+1} \right),$$

следовательно

$$f^{OR}(k) = \left( 1 - (\bar{P}^{k-1})_{1,s+1} \right)^N - \left( 1 - (\bar{P}^k)_{1,s+1} \right)^N.$$

Вариант 2.

Функциональная спецификация ИЛИ-параллельного фрагмента для программы-прототипа 2, выполняющего оценку математического ожидания и дисперсии времени выполнения исследуемого варианта параллельных вычислений, предусматривает два способа нахождения числовых характеристик времени.

Для первого способа целесообразно использовать вычислительные операции с конечным рядом значений времени выполнения параллельной программы и его плотности вероятностей.

Второй способ целесообразно реализовать на основе применения операций линейной алгебры, где исследуемый вариант параллельных вычислений описывается стохастической матрицей вероятностей переходов, определяемой согласно найденной плотности распределения вероятностей времени его выполнения.

### 3.4.2. Альтернативные варианты реализации программного прототипа

#### Вариант 1.

При разработке первого прототипа объединения программ по логической функции «ИЛИ» воспользуемся результатами, полученными в подразделе 3.3. Так как наилучшими показателями обладал третий вариант реализации прототипа, то в данном подразделе воспользуемся схожей моделью граф, которой приведен на рис. 3.19.

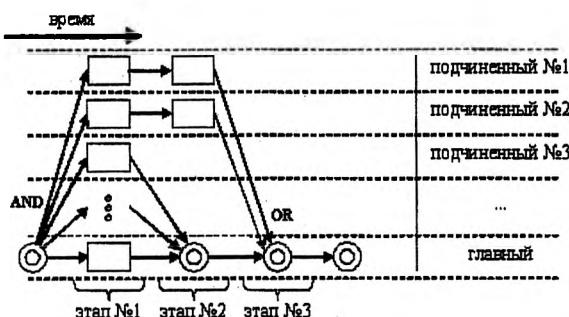


Рис. 3.19. Ориентированный граф сетевой модели параллельной программы для прототипа объединения программ по логической функции «ИЛИ»

Алгоритм работы этого прототипа такой же, как и для реализации третьего варианта в подразделе 3.3.2.

#### Вариант 2.

На рис. 3.20. приведена топология второго варианта прототипа, реализующего вычисления статистических характеристик различными методами.

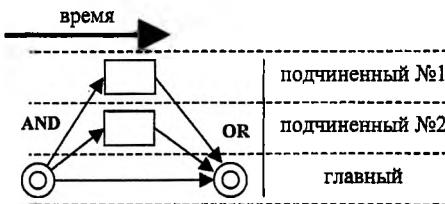


Рис. 3.20. Ориентированный граф модели параллельного прототипа программы для варианта 2

Алгоритм работы:

- основной процесс рассыпает итоговое распределение двум подчиненным процессам;
- оба подчиненных процесса параллельно начинают вычисления математического ожидания и дисперсии различными методами;
- происходит асинхронная передача результата из подчиненного процесса, закончившего вычисления первым;
- происходит прием результата от второго подчиненного процесса.

Вычисление статистических характеристик в первом подчиненном процессе ведется согласно соотношению (3.6). Для второго подчиненного процесса ниже приводятся использующиеся аналитические соотношения.

Анализируемая данным прототипом программа описывается матрицей

$$\bar{P}^{(i)} = \begin{vmatrix} p_{1,1}^{(i)} & p_{1,2}^{(i)} & \dots & p_{1,n}^{(i)} \\ p_{2,1}^{(i)} & p_{2,2}^{(i)} & \dots & p_{2,n}^{(i)} \\ \dots & \dots & \dots & \dots \\ p_{n,1}^{(i)} & p_{n,2}^{(i)} & \dots & p_{n,n}^{(i)} \end{vmatrix}$$

Для элементов матрицы независимо от  $i$  справедливо следующее:

$$p_{i,j} \in [0,1] , \sum_{j=1}^n p_{i,j} = 1.$$

Каноническое представление матрицы независимо от  $i$  имеет вид

$$\bar{P} = \begin{pmatrix} \bar{Q} & \bar{R} \\ \bar{0} & \bar{I} \end{pmatrix},$$

где матрица  $\bar{Q}$  ( $M \times M$ ) – матрица переходов во множество невозвратных состояний;  $\bar{R}$  ( $M \times (n-M)$ ) – матрица переходов из множества невозвратных состояний во множество поглощающих состояний;

$\bar{0}$  ( $(n-M) \times M$ ) – нулевая матрица;

$\bar{I}$  ( $(n-M) \times (n-M)$ ) – единичная матрица.

Определение математических ожиданий осуществляется согласно соотношению:

$$\bar{m} = (\bar{I} - \bar{Q})^{-1} \bar{x}, \quad (3.11)$$

где  $\bar{x}$  – вектор столбец, каждый элемент которого равен 1;

i-ый элемент вектора  $\bar{m}$  равен математическому ожиданию перехода из i-того состояния в конечное.

Для нахождения дисперсий используется формула

$$\bar{d} = [2(\bar{I} - \bar{Q})^{-1} - \bar{I}] \cdot \bar{m} - \bar{m}'' \quad (3.12)$$

где  $\bar{m}_i'' = (\bar{m}_i)^2$ .

Матрица P строится по распределению  $f(n)$   $n=1,2,\dots,N$  следующим образом:

$$P = \begin{pmatrix} f(N) & f(N-1) & f(N-2) & \dots & f(2) & f(1) \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & 0 & \dots & 0 & 1 \end{pmatrix}.$$

### 3.4.3. Методика тестирования и проведение эксперимента

Для тестирования первого варианта прототипа необходимо:

- протестировать работу прототипа на одной исходной матрице;

– протестировать работу прототипа на системе матриц.

Тестируирование второго варианта будет проведено в следующей главе при построении смешанного программного прототипа.

Тестируирование варианта 1.

а) Исходная матрица  $P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$

Для такой матрицы итоговая величина математического ожидания равна 6, а дисперсия равна 0 для любого числа процессов. Следует отметить, что результат не зависит и от точности, поскольку матрица состоит из целочисленных элементов.

б) Исходная матрица  $P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$ .

Для такой матрицы итоговая величина математического ожидания равна 3, а дисперсия равна 0 для любого числа процессов.

Результат работы прототипа при исходных данных а):

```
Введите имя файла содержащего матрицу (*.mtr), or 'q' for quit
#####
## Этап №1 #####
Плотность распределения вер-ти времени выполнения (Процесс 1):
{0.0000000, 0.0000000, 1.0000000, 0.0000000}
Параметр m = 3 (дельта = 0.0000010000).

#####
## Этап №2 #####
Плотность распределения вер-ти времени выполнения (Процесс 2):
{0.0000000, 0.0000000, 1.0000000, 0.0000000}
Параметр m = 3 (дельта = 0.0000010000).

#####
## Этап №3 #####
ОК результат:
{0.0000000, 0.0000000, 0.0000000, 0.0000000, 1.0000000,
 0.0000000}
Среднее = 6.0000, дисперсия = 0.0000
#####
Вычисления завершены ##### (cols = 3)
```

Результат работы прототипа при исходных данных б):

```
##### этап №1 #####
Плотность распределения вер-ти времени выполнения
(Процесс 1):
{0.0000000, 1.0000000, 0.0000000}
параметр m = 2 (дельта = 0.0000010000).

Плотность распределения вер-ти времени выполнения
(Процесс 2):
{0.0000000, 1.0000000, 0.0000000}
параметр m = 2 (дельта = 0.0000010000).

Плотность распределения вер-ти времени выполнения
(Процесс 3):
{0.0000000, 1.0000000, 0.0000000}
параметр m = 2 (дельта = 0.0000010000).

##### Этап №2 #####
##### Этап №3 #####
OR результат:
{0.0000000, 0.0000000, 1.0000000, 0.0000000}
Среднее = 3.0000, дисперсия = 0.0000
##### Вычисления завершены ##### (cols = 3)
```

a) Тест системы матриц

Исходные матрицы:

$$\begin{pmatrix} 0 & 0.2 & 0 & 0 & 0.8 \\ 0 & 0 & 0.2 & 0 & 0.8 \\ 0 & 0 & 0 & 0.2 & 0.8 \\ 0.2 & 0 & 0 & 0 & 0.8 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}_1 \begin{pmatrix} 0 & 0.7 & 0 & 0.3 \\ 0 & 0 & 0.7 & 0.3 \\ 0.5 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix}_2 \begin{pmatrix} 0 & 0.2 & 0 & 0 & 0.8 \\ 0 & 0 & 0.3 & 0 & 0.7 \\ 0 & 0 & 0 & 0.2 & 0.8 \\ 0.3 & 0 & 0 & 0 & 0.7 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}_3$$

$$\begin{pmatrix} 0 & 0.3 & 0 & 0 & 0.7 \\ 0 & 0 & 0.2 & 0 & 0.8 \\ 0 & 0 & 0 & 0.3 & 0.8 \\ 0.2 & 0 & 0 & 0 & 0.8 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}_4 \begin{pmatrix} 0 & 0.5 & 0.5 \\ 0.6 & 0 & 0.4 \\ 0 & 0 & 1 \end{pmatrix}_5.$$

Заданная точность для вычисления:  $\delta = 0.0000001$ .

Полученные результаты совпадают с аналитическими, что подтверждает работоспособность протестированного прототипа.

### 3.5. Программный прототип объединения подпрограмм по логическим функциям «И-ИЛИ»

Часто при реализации параллельных программ объединение происходит по различным логическим функциям, поэтому создание прототипа, в котором будет реализовано совместное использование

различных функций, является вполне закономерным. В реализуемом программном прототипе используются логические функции «И-ИЛИ».

### 3.5.1. Функциональная спецификация прототипа

Спецификация состоит из двух частей: первая аналогична приведенной в подразделе 3.3.1, а вторая – в 3.4.2 для варианта 2. Ориентированный граф, иллюстрирующий сетевую модель реализуемого прототипа, приведен на рис. 3.21.

Результат работы прототипов при исходных данных с):

```
введите имя файла содержащего матрицу (*.mtr), or 'q' for quit
## #####
##### Этап №1 #####
плотность распределения вер-ти времени выполнения (процесс
1):
{0.0000000, 0.8000000, 0.1600000, 0.0320000, 0.0064000,
0.0012800, 0.0002560, 0.0000512, 0.0000102, 0.0000020,
0.000004}
Параметр m = 10 (дельта = 0.0000010000).

плотность распределения вер-ти времени выполнения (процесс
3):
{0.0000000, 0.7000000, 0.2400000, 0.0420000, 0.0144000,
0.0025200, 0.0008640, 0.0001512, 0.0000518, 0.0000091,
0.0000031, 0.0000005}
Параметр m = 11 (дельта = 0.0000010000).

плотность распределения вер-ти времени выполнения (процесс
3):
{0.0000000, 0.5000000, 0.2000000, 0.1500000, 0.0600000,
0.0450000, 0.0180000, 0.0135000, 0.0054000, 0.0040500,
0.0016200, 0.0012150, 0.0004860, 0.0003645, 0.0001458,
0.0001094, 0.0000437, 0.0000328, 0.0000131, 0.0000098
0.0000039, 0.0000030, 0.0000012, 0.0000009, 0.0000004}
Параметр m = 24 (дельта = 0.0000010000).

плотность распределения вер-ти времени выполнения (процесс
2):
{0.0000000, 0.8000000, 0.1400000, 0.0480000, 0.0084000,
0.0028800, 0.0005040, 0.0001728, 0.0000302, 0.0000104,
0.0000018, 0.0000006}
Параметр m = 11 (дельта = 0.0000010000).

плотность распределения вер-ти времени выполнения (процесс
2):
{0.0000000, 0.3000000, 0.2100000, 0.2450000, 0.0735000,
0.0514500, 0.0600250, 0.0180075, 0.0126053, 0.0147061,
0.0044118, 0.0030883, 0.0036030, 0.0010809, 0.0007566,
0.0008827, 0.0002648, 0.0001854, 0.0002163, 0.0000649,
0.0000454, 0.0000530, 0.0000159, 0.0000111, 0.0000130,
0.0000039, 0.0000027, 0.0000032, 0.0000010, 0.0000007,
0.0000008, 0.0000002}
Параметр m = 31 (дельта = 0.0000010000).

#####
##### Этап №2 #####
#####
##### Этап №3 #####
О результаты:
{0.0000000, 0.0000000, 0.8796150, 0.1115057, 0.0084520,
0.0004047, 0.0000203, 0.0000010, 0.0000000, 0.0000000,
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,
0.0000000}

Среднее = 4.2725, дисперсия = 2.7852
#####
Вычисления завершены #####
(co1s = 5)
```

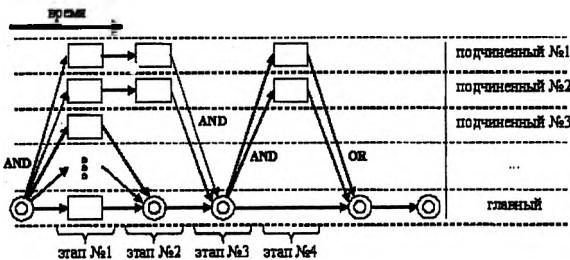


Рис. 3.21. Ориентированный граф сетевой модели прототипа объединения подпрограмм по логическим функциям «И-ИЛИ»

### 3.5.2. Построение схемы измерения временных характеристик

Для измерения временных затрат используется разница между двумя значениями, полученными в результате вызова функции MPI\_Wtime() до и после некоторого события.

Пример кода, иллюстрирующего подобную процедуру:

```
// Логический блок
{
    double rTimeBefore; // время до события
    double rTimeDelta; // временные затраты на событие

    // измеряем текущее время
    rTimeBefore = MPI_Wtime();

    // собственно событие
    ...

    // вычисляем разницу времени - искомые затраты
    rTimeDelta = MPI_Wtime() - rTimeBefore;
}
```

При последовательных измерениях временных затрат возможен другой вариант, применяющийся с целью уменьшения числа дублирующихся вызовов. Сначала в ключевых точках измеряется время, затем данные сохраняются в массиве, который подлежит последующей обработке.

Пример кода, иллюстрирующего подобную процедуру:

```
// Логический блок
{
    double raTime[nNumPoints]; // временные метки
    // измеряем текущее время
    raTime[0] = MPI_Wtime();
```

```

// собственно событие №1
...
// измеряем текущее время
raTime[1] = MPI_Wtime();
...
// измеряем текущее время
raTime[nNumPoints-2] = MPI_Wtime();
// собственно событие №«nNumPoints-1»
...
// измеряем текущее время
raTime[nNumPoints-1] = MPI_Wtime();
// обработка результатов
...
}

```

Временные затраты на вызов функции MPI\_Wtime() и вычисление разницы двух времен можно считать пренебрежимо малыми. Обработка данных проводится в основном процессе.

В качестве меток или точек измерения, позволяющих измерить вычислительные и коммуникационные временные затраты для всех элементов, используются все «узловые» точки программы, т.е. выполняются измерения времени во всех промежутках между вычислениями и коммуникациями, а также в начале и в конце программы.

Следует отметить, что при таком выборе гарантируется достаточно точное измерение времени, затраченного на вычисления. Но в полученном значении времени, затраченном на коммуникации, также учитывается время «простоя» процессов – время ожидания начала приема или передачи сообщений.

### **3.5.3. Методика тестирования, проведение эксперимента и представление результатов**

Поскольку исследуемый программный прототип является частично оттестированным (см. подраздел 3.3.5.), то необходимо проверить ту часть прототипа, которая параллельно вычисляет статистические характеристики (этап 4), обеспечивая объединение процессов по логической функции «ИЛИ». Для этого выявляется зависимость результатов вычислений от количества запущенных процессов и зависимость получаемых результатов от задаваемой точности.

Для эксперимента использованы следующие исходные данные.

Задаваемая точность вычисления:  $\delta = 0.000001$ .

Стохастическая матрица одинакова для всех процессов:

$$\begin{pmatrix} 0 & 0.7 & 0 & 0 & 0 & 0.3 \\ 0 & 0 & 0.3 & 0 & 0 & 0.7 \\ 0 & 0 & 0 & 0.2 & 0 & 0.8 \\ 0 & 0 & 0 & 0 & 0.5 & 0.5 \\ 0.6 & 0 & 0 & 0 & 0 & 0.4 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Результат работы программы (3 процесса):

```
Введите имя файла содержащего матрицу (*.mtr), or 'q' for quit
шу
##### Этап №1 #####
Плотность распределения вер-ти времени выполнения (Процесс
1):
{0.0000000, 0.3000000, 0.4900000, 0.1680000, 0.0210000,
0.0084000, 0.0037800, 0.0061740, 0.0021168, 0.0002646,
0.0001058, 0.0000476, 0.0000778, 0.0000267, 0.0000033,
0.0000013, 0.0000006, 0.0000010, 0.0000003}

Параметр m = 18 (дельта = 0.0000010000).

Плотность распределения вер-ти времени выполнения (Процесс
2):
{0.0000000, 0.3000000, 0.4900000, 0.1680000, 0.0210000,
0.0084000, 0.0037800, 0.0061740, 0.0021168, 0.0002646,
0.0001058, 0.0000476, 0.0000778, 0.0000267, 0.0000033,
0.0000013, 0.0000006, 0.0000010, 0.0000003}

Параметр m = 18 (дельта = 0.0000010000).

Плотность распределения вер-ти времени выполнения (Процесс
3):
{0.0000000, 0.3000000, 0.4900000, 0.1680000, 0.0210000,
0.0084000, 0.0037800, 0.0061740, 0.0021168, 0.0002646,
0.0001058, 0.0000476, 0.0000778, 0.0000267, 0.0000033,
0.0000013, 0.0000006, 0.0000010, 0.0000003}

Параметр m = 18 (дельта = 0.0000010000).

#####
##### Этап №2 #####
#####
##### Этап №3 #####
OR результат:
{0.0000000, 0.0000000, 0.0007290, 0.0558941, 0.3242974,
0.3532921, 0.1394664, 0.0490229, 0.0306786, 0.0262268,
0.0134868, 0.0040574, 0.0012825, 0.0006662, 0.0005135,
0.0002585, 0.0000769, 0.0000240, 0.0000117, 0.0000087,
0.0000043, 0.0000011, 0.0000003, 0.0000001, 0.0000001,
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,
0.0000000, 0.0000000}
```

Процесс, завершивший вычисления первым: 1, ERR = 0, DELTA = 0.000022

Результат: M = 7.109147, D = 3.201824

Процесс, завершивший вычисления вторым: 2, ERR = 0, DELTA = 0.015732

Результат: M = 7.109148, D = 3.201846

Процесс 2, дельта: 0.02508224, тайминг:

{34007.7447416, 34007.7497143, 34007.7551066, 34007.7554351,  
34007.7698174, 34007.7698238}

коммуникации суммарно: 0.01977458, вычисления суммарно:  
0.00530766

Процесс 0, дельта: 0.06439506, тайминг:

{34007.7120595, 34007.7123643, 34007.7702127, 34007.7705046,  
34007.7757394, 34007.7764545}

коммуникации суммарно: 0.06308316, вычисления суммарно:  
0.00131190

Процесс 1, дельта: 0.02093730, тайминг:

{34007.7237423, 34007.7241007, 34007.7342849, 34007.7346328,  
34007.7446762, 34007.7446796}

коммуникации суммарно: 0.02022771, вычисления суммарно:  
0.00070959

В табл. 3.1 приведены результаты, показывающие зависимости времени вычислений и результатов вычислений от количества запущенных процессов. По данным указанной таблицы видно, что второй подпроцесс выполняется дольше первого независимо от запущенного исходного количества процессов.

Таблица 3.1.  
Результаты вычислений

Число процессов	Время на вычисления		Математическое ожидание		Дисперсия	
	подпроцесс №1	подпроцесс №2	подпроцесс №1	подпроцесс №2	подпроцесс №1	подпроцесс №2
3	0.000022	0.015732	7.109147	7.109148	3.201824	3.201846
4	0.000021	0.019860	7.410221	7.410222	3.361192	3.361223
5	0.000021	0.016317	7.650717	7.650718	3.500811	3.500852
6	0.000022	0.016159	7.853134	7.853135	3.620769	3.620821
7	0.000021	0.016145	8.029066	8.029067	3.723249	3.723311

В табл. 3.2 приведены результаты, показывающие зависимости получаемых значений статистических характеристик от точности.

Таблица 3.2.

## Сводная таблица результатов по точности (4 процесса)

Точность	Математическое ожидание		Дисперсия	
	№1	№2	№1	№2
0.01	7.1142601970	7.1378117832	2.6762565355	3.5564296578
0.001	7.3737216702	7.3761005499	3.2026550282	3.2992897861
0.0001	7.4090837416	7.4091437513	3.3527451318	3.3552101070
0.00001	7.4096284287	7.4096584340	3.3562732168	3.3575059312
0.000001	7.4102207723	7.4102215284	3.3611924278	3.3612234984

Из указанной выше таблицы видно, что второй подпроцесс дает результат, приближенный к результату, получаемому при большей точности (см. табл.3.2), однако это не обязательно означает более точный результат по сравнению с подпроцессом №1, т.к. скорее всего, оказывается накапливаемая погрешность матричных операций. Точность влияет на размер получаемого распределения и, следовательно, на размер матрицы в вычислениях математического ожидания и дисперсии подпроцесса №2.

Тексты разработанных прототипов приведены в Приложении 2.

### 3.6. Выводы

1. Разработанные шаблоны программных процедур с функциями библиотеки MPI предназначены для создания программного обеспечения, способного регулировать качество решаемых задач параллельными программами.
2. Проведенный сравнительный анализ построенных прототипов программных процедур выявляет наилучшие варианты с позиции достижения наименьших временных затрат на их выполнение.
3. Построенные системы тестов позволяют определить и продемонстрировать влияние исходных данных на поведение динамических характеристик.

## 4. АНАЛИЗ РАБОТОСПОСОБНОСТИ ПРОТОТИПОВ БАЗОВЫХ КОМПОНЕНТОВ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Для подтверждения корректной работы созданных компонентов необходимо:

- 1) Подтвердить полученные данные при использовании параллельных компонентов результатами работы последовательной программы;
- 2) Подтвердить правильность реализации вычислительных функций аналитическими вычислениями в пакете MathCad.

При разработке последовательных программ для соответствующих компонентов полностью сохранялась их функциональная спецификация. Последовательные программы находятся в тех же проектах что и параллельные. Модификация последовательных программ в параллельные и наоборот зависит от директивы `#define nsCALL_DISABLE_PARALLELING`, имеющейся в каждом проекте.

При запуске параллельных программ исходные данные взяты, как и для тестирования прототипов. Тексты программ для пакета MathCad приведены в Приложении 2.

### 4.1. Подтверждение работоспособности программного прототипа объединения подпрограмм по логической функции «И»

#### 4.1.1. Последовательная программа

Функциональная спецификация программы совпадает с параллельной, их разница состоит только в организации вычислительного процесса.

Ориентированный граф, иллюстрирующий модель последовательной программы для проверки корректной работы прототипов объединения подпрограмм по логической функции «И», показан на рис. 4.1.

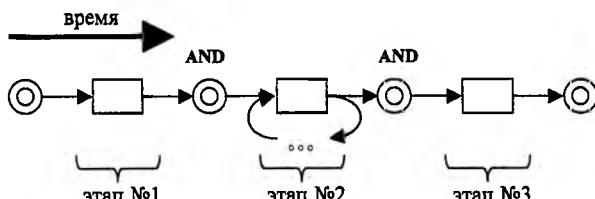


Рис. 4.1. Топология последовательной программы

Результаты работы последовательной программы на данных, приведенных в соответствующих разделах главы 3.

а) Исходная матрица  $P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$ .

Результат работы последовательной программы:

```
Введите имя файла содержащего матрицу (*.mtr), or 'q' for quit
6
##### Этап №1 #####
Плотность распределения вер-ти времени выполнения (Процесс
1):
{0.00000, 0.00000, 1.00000, 0.00000}
Параметр m = 3 (дельта = 0.0000000010).

#####
Плотность распределения вер-ти времени выполнения (Процесс
1):
{0.00000, 0.00000, 1.00000, 0.00000}
Параметр m = 3 (дельта = 0.0000000010).
Свертка (Процесс 1):
{0.00000, 0.00000, 0.00000, 0.00000, 1.00000, 0.00000,
0.00000}
Параметр m = 6 (дельта = 0.0000000010).

#####
Плотность распределения вер-ти времени выполнения (Процесс
1):
{0.00000, 0.00000, 1.00000, 0.00000}
Параметр m = 3 (дельта = 0.0000000010).
Свертка (Процесс 1):
{0.00000, 0.00000, 0.00000, 0.00000, 1.00000, 0.00000,
0.00000}
Параметр m = 6 (дельта = 0.0000000010).

#####
Плотность распределения вер-ти времени выполнения (Процесс
1):
{0.00000, 0.00000, 1.00000, 0.00000}
Параметр m = 3 (дельта = 0.0000000010).
Свертка (Процесс 1):
{0.00000, 0.00000, 0.00000, 0.00000, 1.00000, 0.00000,
0.00000}
Параметр m = 6 (дельта = 0.0000000010).
#####
AND результат:
{0.00000, 0.00000, 0.00000, 0.00000, 1.00000, 0.00000,
0.00000}
Плотность распределения вер-ти времени выполнения (Процесс
1):
{0.00000, 0.00000, 1.00000, 0.00000}
```

Параметр  $m = 3$  (дельта = 0.0000000010).  
 Свертка (Процесс 1):  
 {0.00000, 0.00000, 0.00000, 0.00000, 0.00000, 0.00000,  
 1.00000, 0.00000, 0.00000, 0.00000}  
 Параметр  $m = 9$  (дельта = 0.0000000010).  
 Среднее = 6.0000, Дисперсия = 0.0000  
##### Вычисления завершены #####

б) Исходная матрица  $P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$ .

Результат работы последовательной программы:

```

Введите имя файла содержащего матрицу (*.mtr), or 'q' for quit
3
##### Этап №1 #####
Плотность распределения вер-ти времени выполнения (Процесс
1):
{0.00000, 1.00000, 0.00000}
Параметр  $m = 2$  (дельта = 0.0000000010).

#####
Этап №2 #####
Плотность распределения вер-ти времени выполнения (Процесс
1):
{0.00000, 1.00000, 0.00000}
Параметр  $m = 2$  (дельта = 0.0000000010).
Свертка (Процесс 1):
{0.00000, 0.00000, 1.00000, 0.00000, 0.00000}
Параметр  $m = 4$  (дельта = 0.0000000010).

Плотность распределения вер-ти времени выполнения (Процесс
2):
{0.00000, 1.00000, 0.00000}
Параметр  $m = 2$  (дельта = 0.0000000010).
Свертка (Процесс 2):
{0.00000, 0.00000, 1.00000, 0.00000, 0.00000}
Параметр  $m = 4$  (дельта = 0.0000000010).

#####
Этап №3 #####
AND результат:
{0.00000, 0.00000, 1.00000, 0.00000, 0.00000}
Плотность распределения вер-ти времени выполнения (Процесс
1):
{0.00000, 1.00000, 0.00000}
Параметр  $m = 2$  (дельта = 0.0000000010).
Свертка (Процесс 1):
{0.00000, 0.00000, 0.00000, 1.00000, 0.00000, 0.00000,
0.00000}
Параметр  $m = 6$  (дельта = 0.0000000010).
  
```

Среднее = 3.0000, Дисперсия = 0.0000  
##### Вычисления завершены #####

a) Тест системы матриц.

Исходные матрицы:

$$\begin{pmatrix} 0 & 0.2 & 0 & 0 & 0.8 \\ 0 & 0 & 0.2 & 0 & 0.8 \\ 0 & 0 & 0 & 0.2 & 0.8 \\ 0.2 & 0 & 0 & 0 & 0.8 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}_1 \begin{pmatrix} 0 & 0.7 & 0 & 0.3 \\ 0 & 0 & 0.7 & 0.3 \\ 0.5 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix}_2 \begin{pmatrix} 0 & 0.2 & 0 & 0 & 0.8 \\ 0 & 0 & 0.3 & 0 & 0.7 \\ 0 & 0 & 0 & 0.2 & 0.8 \\ 0.3 & 0 & 0 & 0 & 0.7 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}_3$$
  
$$\begin{pmatrix} 0 & 0.3 & 0 & 0 & 0.7 \\ 0 & 0 & 0.2 & 0 & 0.8 \\ 0 & 0 & 0 & 0.3 & 0.8 \\ 0.2 & 0 & 0 & 0 & 0.8 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}_4 \begin{pmatrix} 0 & 0.5 & 0.5 \\ 0.6 & 0 & 0.4 \\ 0 & 0 & 1 \end{pmatrix}_5$$

Заданная точность для вычисления:  $\delta = 0.0000001$ .

Результат работы последовательной программы:

```
Введите имя файла содержащего матрицу (*.mtr), or 'q' for quit
nn
#####
Этап №1 #####
Плотность распределения вер-ти времени выполнения (Процесс
1):
{0.0000000, 0.8000000, 0.1600000, 0.0320000, 0.0064000,
0.0012800, 0.0002560, 0.0000512, 0.0000102, 0.0000020,
0.0000004}
Параметр m = 10 (дельта = 0.0000010000).

#####
Этап №2 #####
Плотность распределения вер-ти времени выполнения (Процесс
1):
{0.0000000, 0.3000000, 0.2100000, 0.2450000, 0.0735000,
0.0514500, 0.0600250, 0.0180075, 0.0126053, 0.0147061,
0.0044118, 0.0030883, 0.0036030, 0.0010809, 0.0007566,
0.0008827, 0.0002648, 0.0001854, 0.0002163, 0.0000649,
0.0000454, 0.0000530, 0.0000159, 0.0000111, 0.0000130,
0.0000039, 0.0000027, 0.0000032, 0.0000010, 0.0000007,
0.0000008, 0.0000002}
Параметр m = 31 (дельта = 0.0000010000).
Свертка (Процесс 1):
{0.0000000, 0.0000000, 0.2400000, 0.2160000, 0.2392000,
0.1066400, 0.0624880, 0.0605176, 0.0265095, 0.0153861,
0.0148421, 0.0064979, 0.0037702, 0.0036364, 0.0015920,
0.0009237, 0.0008909, 0.0003900, 0.0002263, 0.0002183,
0.0000956, 0.0000554, 0.0000535, 0.0000234, 0.0000136,
0.0000131, 0.0000057, 0.0000033, 0.0000032, 0.0000014,
```

0.0000008, 0.0000008, 0.0000003, 0.0000001, 0.0000000,  
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,  
0.0000000, 0.0000000}

Параметр  $m = 41$  (дельта = 0.0000010000).

Плотность распределения вер-ти времени выполнения (Процесс 1):

{0.0000000, 0.8000000, 0.1400000, 0.0480000, 0.0084000,  
0.0028800, 0.0005040, 0.0001728, 0.0000302, 0.0000104,  
0.0000018, 0.0000006}

Параметр  $m = 11$  (дельта = 0.0000010000).

Свертка (Процесс 1):

{0.0000000, 0.0000000, 0.6400000, 0.2400000, 0.0864000,  
0.0240000, 0.0071040, 0.0018240, 0.0005030, 0.0001248,  
0.0000333, 0.0000081, 0.0000021, 0.0000004, 0.0000001,  
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,  
0.0000000, 0.0000000}

Параметр  $m = 21$  (дельта = 0.0000010000).

Плотность распределения вер-ти времени выполнения (Процесс 1):

{0.0000000, 0.7000000, 0.2400000, 0.0420000, 0.0144000,  
0.0025200, 0.0008640, 0.0001512, 0.0000518, 0.0000091,  
0.0000031, 0.0000005}

Параметр  $m = 11$  (дельта = 0.0000010000).

Свертка (Процесс 1):

{0.0000000, 0.0000000, 0.5600000, 0.3040000, 0.0944000,  
0.0304000, 0.0080960, 0.0023104, 0.0005830, 0.0001581,  
0.0000389, 0.0000103, 0.0000024, 0.0000005, 0.0000001,  
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,  
0.0000000, 0.0000000}

Параметр  $m = 21$  (дельта = 0.0000010000).

##### Этап №3 #####

AND результат:

{0.0000000, 0.0000000, 0.0860160, 0.2606899, 0.2971868,  
0.1413553, 0.0742465, 0.0639962, 0.0275035, 0.0156537,  
0.0149115, 0.0065159, 0.0037746, 0.0036373, 0.0015921,  
0.0009237, 0.0008909, 0.0003900, 0.0002263, 0.0002183,  
0.0000956, 0.0000554, 0.0000535, 0.0000234, 0.0000136,  
0.0000131, 0.0000057, 0.0000033, 0.0000032, 0.0000014,  
0.0000008, 0.0000008, 0.0000003, 0.0000001, 0.0000000,  
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,  
0.0000000, 0.0000000}

Плотность распределения вер-ти времени выполнения (Процесс 1):

{0.0000000, 0.5000000, 0.2000000, 0.1500000, 0.0600000,  
0.0450000, 0.0180000, 0.0135000, 0.0054000, 0.0040500,  
0.0016200, 0.0012150, 0.0004860, 0.0003645, 0.0001458,  
0.0001094, 0.0000437, 0.0000328, 0.0000131, 0.0000098,  
0.0000039, 0.0000030, 0.0000012, 0.0000009, 0.0000004}

Параметр  $m = 24$  (дельта = 0.0000010000).

Свертка (Процесс 1):

```

{0.0000000, 0.0000000, 0.0000000, 0.0430080, 0.1475482,
0.2136338, 0.1743794, 0.1294844, 0.0991612, 0.0653963,
0.0430759, 0.0302054, 0.0191630, 0.0122521, 0.0083225,
0.0051992, 0.0032770, 0.0021900, 0.0013563, 0.0008481,
0.0005613, 0.0003459, 0.0002152, 0.0001416, 0.0000870,
0.0000540, 0.0000354, 0.0000217, 0.0000133, 0.0000087,
0.0000053, 0.0000033, 0.0000021, 0.0000013, 0.0000007,
0.0000004, 0.0000002, 0.0000001, 0.0000001, 0.0000000,
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000}
Параметр m = 65 (дельта = 0.0000010000).

```

Среднее = 6.6313, Дисперсия = 6.9017  
##### Вычисления завершены #####

#### 4.1.2. Построение вычислений в пакете MathCad

Не существует такой матрицы, для которой выполнялось бы  $f_T^{(1)}(k) = 1; k = 0$ ,  $f_T^{(1)}(k) = 0; k \in [1, \infty)$ . (т.е. переход в последнее состояние происходит за 0 шагов), поэтому для сравнения с результатом

аналитической программы задана первая матрица вида:  $\bar{P}^{(1)} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$

(здесь переход в последнее состояние происходит за 1 шаг, учитывая то, что результат для объединения окажется сдвинутым на 1).

Для теста используется следующая матрица:

$$\bar{P}^{(2)} = \begin{pmatrix} 0 & 0.5 & 0 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0.5 & 0.5 \\ 0.5 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Точность – 0.000001, число процессов – 3.

Результат работы параллельной программы:

Плотность распределения вер-ти времени выполнения  
(Процесс 1):  
{0.0000000, 0.5000000, 0.2500000, 0.1250000, 0.0625000,  
0.0312500, 0.0156250, 0.0078125, 0.0039063, 0.0019531,

0.0009766, 0.0004883, 0.0002441, 0.0001221, 0.0000610,  
 0.0000305, 0.0000153, 0.0000076, 0.0000038, 0.0000019,  
 0.0000010, 0.0000005}

AND результат:

{0.0000000, 0.0000000, 0.1250000, 0.2968750, 0.2480469,  
 0.1540527, 0.0851746, 0.0447044, 0.0228915, 0.0115818,  
 0.0058251, 0.0029211, 0.0014627, 0.0007319, 0.0003661,  
 0.0001831, 0.0000915, 0.0000458, 0.0000229, 0.0000114,  
 0.0000057, 0.0000029, 0.0000014, 0.0000000}

Результат работы последовательной программы аналогичен.

Результат работы MathCad:

Плотность распределения:

$\pi^T$	0	1	2	3	4	5	6	
D	0	0	0.5	0.25	0.125	0.063	0.031	0.016

AND результат:

$\pi^T$	0	1	2	3	4	5	6	7	8	9	10	11
D	0	0.125	0.297	0.248	0.154	0.085	0.046	0.023	0.012	0.006	0.003	0.001

Как видим, результат аналитической программы MathCad совпадает с результатом прототипа (результат AND – с точностью до сдвига на 1 элемент).

## 4.2. Подтверждение работоспособности программного прототипа объединения подпрограмм по логической функции «ИЛИ»

### 4.2.1. Последовательная программа

Ориентированный граф, иллюстрирующий модель последовательной программы для проверки корректной работы прототипов объединения подпрограмм по логической функции «ИЛИ», показан на рис. 4.2.

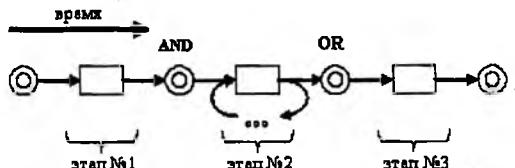


Рис. 4.2. Топология последовательной программы

Результаты работы последовательной программы на данных, приведенных в соответствующих разделах главы 3.

а) Исходная матрица  $P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$ .

Результат работы последовательной программы:

Введите имя файла содержащего матрицу (\*.mtr), or 'q' for quit  
6

##### Этап #1 #####

Плотность распределения вер-ти времени выполнения (Процесс 1):

{0.0000000, 0.0000000, 1.0000000, 0.0000000}

Параметр m = 3 (дельта = 0.0000010000).

##### Этап #2 #####

Плотность распределения вер-ти времени выполнения (Процесс 1):

{0.0000000, 0.0000000, 1.0000000, 0.0000000}

Параметр m = 3 (дельта = 0.0000010000).

Свертка (Процесс 1):

{0.0000000, 0.0000000, 0.0000000, 0.0000000, 1.0000000,  
0.0000000, 0.0000000}

Параметр m = 6 (дельта = 0.0000010000).

.....

Среднее = 6.0000, Дисперсия = 0.0000

##### Вычисления завершены ##### (cols = 3)

б) Исходная матрица  $P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$ .

Результат работы последовательной программы:

Введите имя файла содержащего матрицу (\*.mtr), or 'q' for quit  
3

##### Этап #1 #####

Плотность распределения вер-ти времени выполнения (Процесс 1):

{0.0000000, 1.0000000, 0.0000000}

Параметр m = 2 (дельта = 0.0000010000).

Плотность распределения вер-ти времени выполнения (Процесс 2):  
{0.0000000, 1.0000000, 0.0000000}  
Параметр m = 2 (дельта = 0.0000010000).

Плотность распределения вер-ти времени выполнения (Процесс 3):  
{0.0000000, 1.0000000, 0.0000000}  
Параметр m = 2 (дельта = 0.0000010000).

##### Этап №2 #####  
##### Этап №3 #####  
OR результат:  
{0.0000000, 0.0000000, 1.0000000, 0.0000000}  
Среднее = 3.0000, Дисперсия = 0.0000  
##### Вычисления завершены ##### (cols = 3)

### a) Тест системы матриц.

Исходные матрицы:

$$\begin{pmatrix} 0 & 0.2 & 0 & 0 & 0.8 \\ 0 & 0 & 0.2 & 0 & 0.8 \\ 0 & 0 & 0 & 0.2 & 0.8 \\ 0.2 & 0 & 0 & 0 & 0.8 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}_1 \begin{pmatrix} 0 & 0.7 & 0 & 0.3 \\ 0 & 0 & 0.7 & 0.3 \\ 0.5 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix}_2 \begin{pmatrix} 0 & 0.2 & 0 & 0 & 0.8 \\ 0 & 0 & 0.3 & 0 & 0.7 \\ 0 & 0 & 0 & 0.2 & 0.8 \\ 0.3 & 0 & 0 & 0 & 0.7 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}_3$$
  
$$\begin{pmatrix} 0 & 0.3 & 0 & 0 & 0.7 \\ 0 & 0 & 0.2 & 0 & 0.8 \\ 0 & 0 & 0 & 0.3 & 0.8 \\ 0.2 & 0 & 0 & 0 & 0.8 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}_4 \begin{pmatrix} 0 & 0.5 & 0.5 \\ 0.6 & 0 & 0.4 \\ 0 & 0 & 1 \end{pmatrix}_5$$

Заданная точность для вычисления:  $\delta = 0.0000001$ .

Результат работы последовательной программы:

Введите имя файла содержащего матрицу (\*.mtr), or 'q' for quit  
пп

##### Этап №1 #####

Плотность распределения вер-ти времени выполнения (Процесс 1):

{0.0000000, 0.8000000, 0.1600000, 0.0320000, 0.0064000,  
0.0012800, 0.0002560, 0.0000512, 0.0000102, 0.0000020,  
0.0000004}

Параметр m = 10 (дельта = 0.0000010000).

##### Этап №2 #####

Плотность распределения вер-ти времени выполнения (Процесс 1):

{0.0000000, 0.3000000, 0.2100000, 0.2450000, 0.0735000,  
0.0514500, 0.0600250, 0.0180075, 0.0126053, 0.0147061,  
0.0044118, 0.0030883, 0.0036030, 0.0010809, 0.0007566,

0.0008827, 0.0002648, 0.0001854, 0.0002163, 0.0000649,  
0.0000454, 0.0000530, 0.0000159, 0.0000111, 0.0000130,  
0.0000039, 0.0000027, 0.0000032, 0.0000010, 0.0000007,  
0.0000008, 0.0000002}

Параметр  $m = 31$  (дельта = 0.0000010000).

Свертка (Процесс 1):

{0.0000000, 0.0000000, 0.2400000, 0.2160000, 0.2392000,  
0.1066400, 0.0624880, 0.0605176, 0.0265095, 0.0153861,  
0.0148421, 0.0064979, 0.0037702, 0.0036364, 0.0015920,  
0.0009237, 0.0008909, 0.0003900, 0.0002263, 0.0002183,  
0.0000956, 0.0000554, 0.0000535, 0.0000234, 0.0000136,  
0.0000131, 0.0000057, 0.0000033, 0.0000032, 0.0000014,  
0.0000008, 0.0000008, 0.0000003, 0.0000001, 0.0000000,  
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,  
0.0000000, 0.0000000}

Параметр  $m = 41$  (дельта = 0.0000010000).

Плотность распределения вер-ти времени выполнения (Процесс 1):

{0.0000000, 0.8000000, 0.1400000, 0.0480000, 0.0084000,  
0.0028800, 0.0005040, 0.0001728, 0.0000302, 0.0000104,  
0.0000018, 0.0000006}

Параметр  $m = 11$  (дельта = 0.0000010000).

Свертка (Процесс 1):

{0.0000000, 0.0000000, 0.6400000, 0.2400000, 0.0864000,  
0.0240000, 0.0071040, 0.0018240, 0.0005030, 0.0001248,  
0.0000333, 0.0000081, 0.0000021, 0.0000004, 0.0000001,  
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,  
0.0000000, 0.0000000}

Параметр  $m = 21$  (дельта = 0.0000010000).

Плотность распределения вер-ти времени выполнения (Процесс 1):

{0.0000000, 0.7000000, 0.2400000, 0.0420000, 0.0144000,  
0.0025200, 0.0008640, 0.0001512, 0.0000518, 0.0000091,  
0.0000031, 0.0000005}

Параметр  $m = 11$  (дельта = 0.0000010000).

Свертка (Процесс 1):

{0.0000000, 0.0000000, 0.5600000, 0.3040000, 0.0944000,  
0.0304000, 0.0080960, 0.0023104, 0.0005830, 0.0001581,  
0.0000389, 0.0000103, 0.0000024, 0.0000005, 0.0000001,  
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,  
0.0000000, 0.0000000}

Параметр  $m = 21$  (дельта = 0.0000010000).

##### Этап №3 #####

OR результат:

{0.0000000, 0.0000000, 0.8796150, 0.1115057, 0.0084520,  
0.0004047, 0.0000203, 0.0000010, 0.0000000, 0.0000000,  
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,  
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,  
0.0000000}

Плотность распределения вер-ти времени выполнения (Процесс 1):

{0.0000000, 0.5000000, 0.2000000, 0.1500000, 0.0600000,  
 0.0450000, 0.0180000, 0.0135000, 0.0054000, 0.0040500,  
 0.0016200, 0.0012150, 0.0004860, 0.0003645, 0.0001458,  
 0.0001094, 0.0000437, 0.0000328, 0.0000131, 0.0000098,  
 0.0000039, 0.0000030, 0.0000012, 0.0000009, 0.0000004}

Параметр  $m = 24$  (дельта = 0.0000010000).

Свертка (Процесс 1):

{0.0000000, 0.0000000, 0.0000000, 0.4398075, 0.2316759,  
 0.1584694, 0.0713955, 0.0476319, 0.0214232, 0.0142898,  
 0.0064270, 0.0042869, 0.0019281, 0.0012861, 0.0005784,  
 0.0003858, 0.0001735, 0.0001157, 0.0000521, 0.0000347,  
 0.0000156, 0.0000104, 0.0000047, 0.0000031, 0.0000014,  
 0.0000009, 0.0000004, 0.0000000, 0.0000000, 0.0000000,  
 0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,  
 0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,  
 0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000}

Параметр  $m = 44$  (дельта = 0.0000010000).

Среднее = 4.2725, Дисперсия = 2.7852

##### Вычисления завершены ##### (cols = 5)

#### 4.2.2. Построение вычислений в пакете MathCad

Для теста используется следующая матрица:

$$\bar{P}^{(2)} = \begin{pmatrix} 0 & 0.5 & 0 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0.5 & 0.5 \\ 0.5 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Точность – 0.000001, число процессов – 3.

Результат работы параллельной программы:

Плотность распределения вер-ти времени выполнения  
 (Процесс 1):

{0.0000000, 0.5000000, 0.2500000, 0.1250000, 0.0625000,  
 0.0312500, 0.0156250, 0.0078125, 0.0039063, 0.0019531,  
 0.0009766, 0.0004883, 0.0002441, 0.0001221, 0.0000610,  
 0.0000305, 0.0000153, 0.0000076, 0.0000038, 0.0000019,  
 0.0000010, 0.0000005}

AND результат:

{0.0000000, 0.0000000, 0.8751200, 0.1091000, 0.0143400,  
 0.0021372, 0.0000000, 0.0000000, 0.0000000, 0.0000000,  
 0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,  
 0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,  
 0.0000000, 0.0000000, 0.0000000, 0.0000000}

Результат работы последовательной программы аналогичен.

Результат работы MathCad.

Плотность распределения:

	0	0.5	0.25	0.125	0.063	0.031
0	0	0.5	0.25	0.125	0.063	0.031

Результат при функции «ИЛИ»:

	0	0.5	0.25	0.125	0.063	0.031
0	0	0.875	0.109	0.014	0.002	0

Как видим, результат аналитической программы MathCad совпадает с результатом прототипа с точностью до сдвига на 1 элемент.

#### 4.3. Выводы

1. Согласование результатов выполнения созданного программного средства с данными, полученными с применением пакета MathCad, подтверждает работоспособность программного прототипа объединения подпрограмм по логической функции «И».
2. Согласование результатов выполнения созданного программного средства с данными, полученными с применением пакета MathCad, подтверждает работоспособность программного прототипа объединения подпрограмм по логической функции «ИЛИ».

## **5. ОЦЕНКА СТАТИСТИЧЕСКИХ ХАРАКТЕРИСТИК ОБНАРУЖЕНИЯ И ОТРАЖЕНИЯ УГРОЗ ДЛЯ АВТОМАТИЗИРОВАННЫХ СИСТЕМ ТРЕТЬЕГО КЛАССА ЗАЩИЩЕННОСТИ В БАЗИСЕ ФУНКЦИЙ МРІ**

### **5.1. Базовая модель процессов обнаружения и отражения угроз для автоматизированных систем третьего класса защищенности**

В связи с расширением применения автоматизированных систем (АС), подлежащих защите от несанкционированного доступа к информации, возникла актуальная задача разработки их моделей в условиях воздействия угроз. Действующие в настоящее время в РФ нормы по защите информации от несанкционированного доступа представлены в руководящих документах Государственной технической комиссии и стандартах [37 – 42]. Стандарты определяют две группы критериев информационной безопасности – показатели защищенности средств вычислительной техники от несанкционированного доступа и критерии защищенности автоматизированных систем. Первая группа позволяет оценить степень защищенности отдельно поставляемых потребителю компонентов вычислительных систем, а вторая рассчитана на полнофункциональные системы обработки данных. Классификация требований к средствам защиты информации (СЗИ) автоматизированных систем от несанкционированного доступа приведена на рис.5.1.

Данные требования являются составной частью критериев защищенности АС от несанкционированного доступа к информации (НСД). Требования сгруппированы вокруг реализующих их подсистем защиты. К недостаткам содержания рассматриваемого документа относится отсутствие требований к СЗИ по качеству их функционирования, а также к адекватности реализации политики безопасности. Частным аспектом указанных недостатков СЗИ является отсутствие требований по управлению процессами защиты информации. В связи с этим возникает возможность реализации различных вариантов построения систем и неопределенность выбора наилучшего способа их организации. Для снятия подобной неопределенности необходимо построить модель СЗИ, которая позволит находить их характеристики в зависимости от архитектурных особенностей. В общем случае временные характеристики определяются по отношению к некоторой группе событий. Временные характеристики находятся на основе использования логической модели вычислительного процесса. Полученные результаты рассматриваются при сравнении различных структур СЗИ. При этом

руководствуются оценками вероятностно-временного характера, имеющими смысл плотностей и функций распределения [32, 33, 34]. В частности, к ним относится и вероятность преодоления системы защиты информации за некоторое время, которая используется в качестве показателя риска.

Согласно требованиям руководящих документов для автоматизированных систем третьего класса защищенности построена логическая модель, приведенная на рис. 5.2. Модель обнаружения и отражения угрозы однозначно зависит от типа угрозы, поэтому для конкретной реализации угрозы можно определить путь ее прохождения внутри системы. Базовая модель обнаружения построена для обобщенной угрозы, которая сочетает все виды возможных воздействий на комплекс средств защиты информации.

Обобщенная угроза может быть обнаружена различными механизмами безопасности, которые работают либо последовательно, либо параллельно. В случае параллельной работы механизмов безопасности угроза может быть обнаружена быстрее (объединение по функции «ИЛИ» (OR)). Отражение угрозы есть результат объединения по функции «И» (AND) (угроза считается отраженной, когда ее отразили все средства защиты).

Следует отметить, что в случае возникновения потребности в анализе конкретного типа угроз, необходимо строить модели конкретных реализаций угроз. Найденные динамические характеристики для комбинации моделей позволят оценить эффективность защиты от данного типа угроз.

В представленной модели подсистема управления доступом и подсистема контроля целостности работают параллельно. Для подсистемы управления доступом угроза может быть не отражена, отражена, отражена и обнаружена. Для подсистемы обеспечения целостности определены следующие выходные состояния: угроза не отражена, угроза обнаружена, но неотраженна, угроза обнаружена и отражена.

Указанная модель может быть представлена как графически (рис.5.2), так и совокупностью матрицы переходных вероятностей и множеством, задающим плотности вероятностей времени выполнения соответствующих подпроцессов. Так как матрица переходов для данной модели получается сильно разреженной, то целесообразно ее представить в виде списков переходов (табл. 5.1).



Рис. 5.1. Требования к средствам защиты АС от несанкционированного доступа к информации

Таблица 5.1.

## Переходы графа модели

Переход	Вероятность перехода	Переход	Вероятность перехода	Переход	Вероятность перехода
1→2	1	12→21	1	25→35	1
1→3	1	16→21	1	26→36	1
2→3	Pid1	16→22	1	27→37	1
2→4	Pid2	13→22	1	28→38	1
2→5	1-Pid1-Pid2	17→23	1	29→39	1
3→6	Pint1	12→23	1	30→40	1
3→7	Pint2	17→24	1	31→40	1
3→8	1-Pint1-Pint2	13→24	1	32→40	1
3→45	1	10→25	1	33→40	1
9→45	1	13→25	1	34→41	1
45→16	Preact1	18→26	1	35→41	1
45→17	1-Preact1	14→26	1	36→42	1
4→9	Preg1	19→27	1	37→43	1
4→10	1-Preg1	15→27	1	38→44	1
5→11	Preg2	10→28	1	39→44	1
5→47	1-Preg2	8→28	1		
6→12	Preact3	8→29	1		
6→13	1-Preact3	47→29	1		
11→46	1	20→30	1		
46→18	Preact2	21→31	1		
46→19	1-Preact2	22→32	1		
12→20	1	23→33	1		
10→20	1	24→34	1		

Представление исходных характеристик для базовой модели процессов обнаружения и отражения угроз для автоматизированных систем третьего класса защищенности показано в табл.5.2.

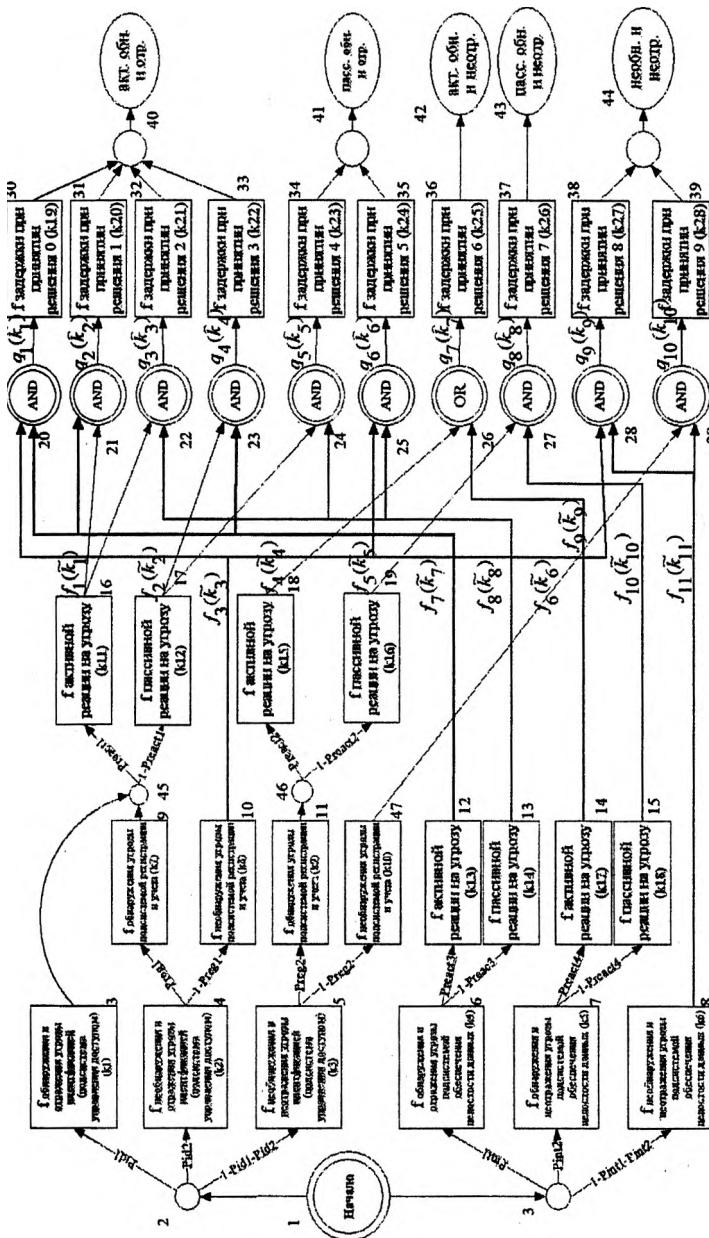


Рис. 5.2. Базовая модель процессов обнаружения и отражения угроз для автоматизированных систем третьего класса защиты



Таблица 5.2.

## Исходные характеристики модели

Обозначение объекта	Смыслоное содержание объекта
$f_{\text{обн.отр.ид}}(k_1)$ $k_1=1, \dots, k_{\text{обн.отр.ид}}$	Плотность распределения времени обнаружения и отражения угрозы при идентификации (подсистема управления доступом)
$f_{\text{необн.отр.ид}}(k_2)$ $k_2=1, \dots, k_{\text{необн.отр.ид}}$	Плотность распределения времени необнаружения и отражения угрозы при идентификации (подсистема управления доступом)
$f_{\text{необн.неотр.ид}}(k_3)$ $k_3=1, \dots, k_{\text{необн.неотр.ид}}$	Плотность распределения времени необнаружения и неотражения угрозы при идентификации (подсистема управления доступом)
$f_{\text{обн.отр.цел}}(k_4)$ $k_4=1, \dots, k_{\text{обн.отр.цел}}$	Плотность распределения времени обнаружения и отражения угрозы подсистемой обеспечения целостности данных
$f_{\text{обн.неотр.цел}}(k_5)$ $k_5=1, \dots, k_{\text{обн.неотр.цел}}$	Плотность распределения времени обнаружения и неотражения угрозы подсистемой обеспечения целостности данных
$f_{\text{необн.неотр.цел}}(k_6)$ $k_6=1, \dots, k_{\text{необн.неотр.цел}}$	Плотность распределения времени необнаружения и неотражения угрозы подсистемой обеспечения целостности данных
$f_{\text{обн.рег.уч}}(k_7)$ $k_7=1, \dots, k_{\text{обн.рег.уч}}$	Плотность распределения времени обнаружения угрозы подсистемой регистрации и учета при необнаружении и отражении угрозы подсистемой управления доступом
$f_{\text{необн.рег.уч}}(k_8)$ $k_8=1, \dots, k_{\text{необн.рег.уч}}$	Плотность распределения времени необнаружения угрозы подсистемой регистрации и учета при необнаружении и отражении угрозы подсистемой управления доступом
$f_{\text{обн.рег.уч}}(k_9)$ $k_9=1, \dots, k_{\text{обн.рег.уч}}$	Плотность распределения времени обнаружения угрозы подсистемой регистрации и учета при необнаружении и неотражении угрозы подсистемой управления доступом

$f_{\text{необр.рег.уч}}(k_{10})$ $k_{10}=1, \dots, k_{\text{необр.рег.уч}}$	Плотность распределения времени необнаружения угрозы подсистемой регистрации и учета при необнаружении и неотражении угрозы подсистемой управления доступом
$f_{\text{акт.реак.}}(k_{11})$ $k_{11}=1, \dots, k_{\text{акт.реак.}}$	Плотность распределения времени активной реакции на угрозу при обнаружении угрозы подсистемой управления доступом или подсистемой регистрации и учета
$f_{\text{пассив.реак.}}(k_{12})$ $k_{12}=1, \dots, k_{\text{пассив.реак.}}$	Плотность распределения времени пассивной реакции на угрозу при обнаружении угрозы подсистемой управления доступом или подсистемой регистрации и учета
$f_{\text{акт.реак.}}(k_{13})$ $k_{13}=1, \dots, k_{\text{акт.реак.2}}$	Плотность распределения времени активной реакции на угрозу при необнаружении угрозы подсистемой управления доступом и обнаружении подсистемой регистрации и учета
$f_{\text{пассив.реак.}}(k_{14})$ $k_{14}=1, \dots, k_{\text{пассив.реак.2}}$	Плотность распределения времени пассивной реакции на угрозу при необнаружении угрозы подсистемой управления доступом и обнаружении подсистемой регистрации и учета
$f_{\text{акт.реак.}}(k_{15})$ $k_{15}=1, \dots, k_{\text{акт.реак.3}}$	Плотность распределения времени активной реакции на угрозу при обнаружении и отражении подсистемой обеспечения целостности данных
$f_{\text{пассив.реак.}}(k_{16})$ $k_{16}=1, \dots, k_{\text{пассив.реак.3}}$	Плотность распределения времени пассивной реакции на угрозу при обнаружении и отражении подсистемой обеспечения целостности данных
$f_{\text{акт.реак.}}(k_{17})$ $k_{17}=1, \dots, k_{\text{акт.реак.4}}$	Плотность распределения времени активной реакции на угрозу при обнаружении и неотражении подсистемой обеспечения целостности данных
$f_{\text{пассив.реак.}}(k_{18})$ $k_{18}=1, \dots, k_{\text{пассив.реак.4}}$	Плотность распределения времени пассивной реакции на угрозу при обнаружении и неотражении подсистемой обеспечения целостности данных
$f_{\text{зад.реш.}}(k_{19}), \dots,$ $f_{\text{зад.реш.}}(k_{28})$ $k_{19}=1, \dots, k_{\text{зад.реш.5}}$ $\dots; k_{28}=1, \dots, k_{\text{зад.реш.9}}$	Плотность распределения времени задержки при принятии решения I, I=0,1,..,9

Обозначения динамических характеристик комплекса средств защиты информации для автоматизированных систем третьего класса защищенности приведены в табл. 5.3.

Таблица 5.3  
Обозначения динамических характеристик

Плотности распределения для каждого итогового решения	Смыслоное содержание
$f_{акт.обн.отр.}(k_1)$ $k_1=1, \dots, k_{акт.обн.отр}$	Плотность распределения времени активного обнаружения и отражения атаки
$f_{пассив.обн.отр.}(k_2)$ $k_2=1, \dots, k_{пассив.обн.отр}$	Плотность распределения времени пассивного обнаружения и отражения атаки
$f_{акт.обн.неотр.}(k_3)$ $k_3=1, \dots, k_{акт.обн.неотр}$	Плотность распределения времени активного обнаружения и неотражения атаки
$f_{пассив.обн.неотр.}(k_4)$ $k_4=1, \dots, k_{пассив.обн.неотр}$	Плотность распределения времени пассивного обнаружения и неотражения атаки
$f_{необн.неотр.}(k_5)$ $k_5=1, \dots, k_{необн.неотр}$	Плотность распределения времени необнаружения и неотражения атаки

## 5.2. Вывод аналитических соотношений для определения динамических характеристик

Для определения характеристик по построенной модели потребуются соотношения, получаемые с использованием базовых функций метода свертки.

Для последовательных процессов в составе параллельных, связанных по схеме с рис.5.3, плотность распределения времени их окончания определяется по соотношению (5.1).

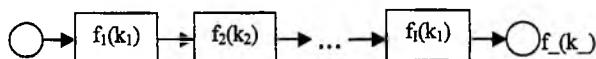


Рис. 5.3. Последовательные процессы в составе параллельных

Плотность распределения времени их окончания определяется следующим образом:

$$f_{12..j}(k_{12..j}) = \sum_{k_{12..(j-1)}} f_{12..(j-1)}(k_{12..(j-1)}) \cdot f_j(k_{12..j} - k_{12..(j-1)}),$$

$$k_{12..j} = \min(k_1 + k_2 + \dots + k_j), \dots, \max(k_1 + k_2 + \dots + k_j),$$

$$j = 2, \dots, I;$$

$$f_{\_}(k\_)=f_{12..I}(k_{12..I}=k\_); \quad (5.1)$$

$$k\_ = \min(k_1, k_{12}, \dots, k_{12..I}), \dots, \max(k_1, k_{12}, \dots, k_{12..I}),$$

где  $f_{12..j}(k_{12..j})$  – плотность распределения вероятности времени окончания  $j$ -ой последовательной программы,  $f_{\_}(k\_)$  – плотность распределения вероятности времени окончания последовательного процесса в составе параллельного,  $k\_$  – время окончания выполнения последовательного процесса.

Для параллельных процессов, объединяемых по логической функции «И» (рис.5.4.), плотность распределения времени их окончания определяется соотношением (5.2).

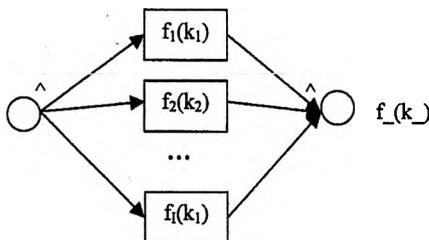


Рис. 5.4. Параллельные процессы при объединении по логической функции «И»

$$f_{\_}(k\_)=f_1(k_1=k\_)\sum_{k_2\leq k\_}\dots\sum_{k_I\leq k\_}f_2(k_2)\cdot\dots\cdot f_I(k_I)+$$

$$+f_2(k_2=k\_)\sum_{k_1< k\_}\sum_{k_3\leq k\_}\dots\sum_{k_I\leq k\_}f_1(k_1)\cdot f_3(k_3)\cdot\dots\cdot f_I(k_I)+$$

$$+\dots+f_I(k_I=k\_)\sum_{k_1< k\_}\dots\sum_{k_{I-1}< k\_}f_1(k_1)\cdot\dots\cdot f_{I-1}(k_{I-1}) \quad (5.2)$$

$$k\_ = \max_i(\min k_1, \min k_2, \dots, \min k_I), \dots, \max_i(\max k_1, \max k_2, \dots, \max k_I),$$

где  $f_i(k_i)$  – плотность распределения вероятности  $k_i$  времени окончания параллельного процесса,  $k_i$  – дискретная величина, отображающая время окончания выполнения последовательной программы в составе параллельной,  $i = 1, 2, \dots, I$ .

Для параллельных процессов объединяемых по логической функции «ИЛИ» (рис. 5.5). плотность распределения времени их окончания определяется по формулам (5.3):

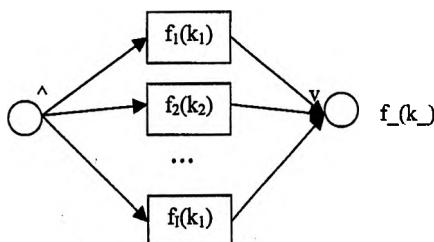


Рис. 5.5. Параллельные процессы при объединении по логической функции «ИЛИ»

$$\begin{aligned}
 f_{\text{--}}(k_{\text{--}}) &= f_1(k_1 = k_{\text{--}}) \sum_{k_2 \geq k_{\text{--}}} \dots \sum_{k_I \geq k_{\text{--}}} f_2(k_2) \dots f_I(k_I) + \\
 &+ f_2(k_2 = k_{\text{--}}) \sum_{k_1 > k_{\text{--}}} \sum_{k_3 \geq k_{\text{--}}} \dots \sum_{k_I \geq k_{\text{--}}} f_1(k_1) \cdot f_3(k_3) \dots f_I(k_I) + \\
 &+ \dots + f_I(k_I = k_{\text{--}}) \sum_{k_1 > k_{\text{--}}} \dots \sum_{k_{I-1} > k_{\text{--}}} f_1(k_1) \dots f_{I-1}(k_{I-1})
 \end{aligned} \tag{5.3}$$

$$k_{\text{--}} = \min_i(\min k_i, \max k_i), \dots, \min_i(\max k_i, \max k_i),$$

где  $f_i(k_i)$  – плотность распределения вероятности  $k_i$  времени окончания параллельного процесса,  $k_i$  – дискретная величина, отображающая время окончания выполнения последовательной программы в составе параллельной,  $i = 1, 2, \dots, I$ .

Для большей наглядности выводится 11 промежуточных выражений для  $f_1(\cdot), \dots, f_{11}(\cdot)$ , предшествующих объединению подграфов по логическим функциям.

Первая величина представляет собой плотность распределения времени активной реакции на угрозу, обнаруженную и отраженную подсистемой управления доступом либо обнаруженную подсистемой регистрации и учета при необнаружении и отражении угрозы подсистемой управления доступом:

$$\begin{aligned}
f_1(\tilde{k}_1) = & p_{id1} \cdot p_{react.1} \sum_{k'} f_{обн.отр..ид}(k') \cdot f_{акт..реакц.}(\tilde{k}_1 - k') + p_{id2} \cdot p_{reg.1} \cdot p_{react.1} \cdot \\
& \cdot \sum_{k''} (\sum_{k'} f_{необн.отр..ид}(k'') \cdot f_{обн.рег.уч.}(k''' - k'')) \cdot f_{акт..реакц.}(\tilde{k}_1 - k'''); \\
k' = & k_1; k_1 = 1, \dots, k_{обн.отр..ид}; k' = 1, \dots, k_{обн.отр..ид}; \\
k'' = & k_2; k_2 = 1, \dots, k_{необн.отр..ид}; k'' = 1, \dots, k_{необн.отр..ид}; \\
k''' = & k_2 + k_7; k_7 = 1, \dots, k_{обн.рег.уч.}; k''' = 2, \dots, k_{необн.отр..ид} + k_{обн..рег..уч.}; \\
\tilde{k}_1 = & \min(k_1 + k_{11}, k_2 + k_7 + k_{11}), \dots, \max(k_1 + k_{11}, k_2 + k_7 + k_{11}); \\
k_{11} = & 1, \dots, k_{акт..реакц.1}.
\end{aligned} \tag{5.4}$$

Вторая величина это плотность распределения времени пассивной реакции на угрозу, обнаруженную и отраженную подсистемой управления доступом либо обнаруженную подсистемой регистрации и учета при необнаружении и отражении угрозы подсистемой управления доступом:

$$\begin{aligned}
f_2(\tilde{k}_2) = & p_{id1} \cdot (1 - p_{react.1}) \sum_{k'} f_{обн.отр..ид}(k') \cdot f_{насс.реакц.}(\tilde{k}_2 - k') + p_{id2} \cdot p_{reg.1} \cdot (1 - p_{react.1}) \cdot \\
& \cdot \sum_{k''} (\sum_{k'} f_{необн.отр..ид}(k'') \cdot f_{обн.рег.уч.}(k''' - k'')) \cdot f_{насс..реакц.}(\tilde{k}_2 - k'''); \\
k' = & k_1; k_1 = 1, \dots, k_{обн.отр..ид}; k' = 1, \dots, k_{обн.отр..ид}; \\
k'' = & k_2; k_2 = 1, \dots, k_{необн.отр..ид}; k'' = 1, \dots, k_{необн.отр..ид}; \\
k''' = & k_2 + k_7; k_7 = 1, \dots, k_{обн.рег.уч.}; k''' = 2, \dots, k_{необн.отр..ид} + k_{обн..рег..уч.}; \\
\tilde{k}_2 = & \min(k_1 + k_{12}, k_2 + k_7 + k_{12}), \dots, \max(k_1 + k_{12}, k_2 + k_7 + k_{12}); \\
k_{12} = & 1, \dots, k_{насс.реакц.1}.
\end{aligned} \tag{5.5}$$

Третья величина есть плотность распределения времени необнаружения угрозы подсистемой регистрации и учета при необнаружении и отражении угрозы подсистемой управления доступом:

$$\begin{aligned}
f_3(\tilde{k}_3) = & p_{id2} \cdot (1 - p_{reg.1}) \sum_{k'} f_{необн..отр..ид}(k') \cdot f_{необр..отр..ид.}(\tilde{k}_3 - k'); \\
k' = & k_2; k_2 = 1, \dots, k_{необн..отр..ид}; k' = 1, \dots, k_{необн..отр..ид}; \\
\tilde{k}_3 = & 2, \dots, (k_2 + k_8); \\
k_8 = & 1, \dots, k_{необр..рег..уч.}.
\end{aligned} \tag{5.6}$$

Четвертая величина представляет собой плотность распределения времени активной реакции на угрозу, обнаруженную подсистемой

регистрации и учета при необнаружении и неотражении угрозы подсистемой управления доступом:

$$f_4(\tilde{k}_4) = (1 - p_{id1} - p_{id2}) \cdot p_{reg2} \cdot p_{react2} \sum_{k''} \left( \sum_{k'} f_{\text{необр.неотр.ид}}(k') \cdot f_{\text{обр.рег.уч}}(k'' - k') \right) \cdot \\ \cdot f_{\text{акт.реакц}}(\tilde{k}_4 - k''); \\ k' = k_3; k_3 = 1,..,k_{\text{необр.неотр.ид}}; k'' = 1,..,k_{\text{необр.неотр.ид}}; \\ k'' = k_3 + k_9; k_9 = 1,..,k_{\text{обр.рег.уч}}; k'' = 2,..,k_{\text{необр.неотр.ид}} + k_{\text{обр.рег.уч}}; \\ \tilde{k}_4 = 2,..,(k_3 + k_9 + k_{13}); \\ k_{13} = 1,..,k_{\text{акт.реакц2}}. \quad (5.7)$$

Пятая величина представляет собой плотность распределения времени пассивной реакции на угрозу, обнаруженную подсистемой регистрации и учета при необнаружении и неотражении угрозы подсистемой управления доступом:

$$f_5(\tilde{k}_5) = (1 - p_{id1} - p_{id2}) \cdot p_{reg2} \cdot (1 - p_{react2}) \sum_{k''} \left( \sum_{k'} f_{\text{необр.неотр.ид}}(k') \cdot f_{\text{обр.рег.уч}}(k'' - k') \right) \cdot \\ \cdot f_{\text{пассив.реакц}}(\tilde{k}_5 - k''); \\ k' = k_3; k_3 = 1,..,k_{\text{необр.неотр.ид}}; k'' = 1,..,k_{\text{необр.неотр.ид}}; \\ k'' = k_3 + k_9; k_9 = 1,..,k_{\text{обр.рег.уч}}; k'' = 2,..,k_{\text{необр.неотр.ид}} + k_{\text{обр.рег.уч}}; \\ \tilde{k}_5 = 3,..,(k_3 + k_9 + k_{14}); \\ k_{14} = 1,..,k_{\text{пассив.реакц2}}. \quad (5.8)$$

Шестая величина есть плотность распределения времени необнаружения угрозы подсистемой регистрации и учета при необнаружении и неотражении угрозы подсистемой управления доступом:

$$f_6(\tilde{k}_6) = (1 - p_{id1} - p_{id2}) \cdot (1 - p_{reg2}) \sum_k f_{\text{необр.неотр.ид}}(k') \cdot f_{\text{необр.рег.уч}}(\tilde{k}_6 - k'); \\ k' = k_3; k_3 = 1,..,k_{\text{необр.неотр.ид}}; k' = 1,..,k_{\text{необр.неотр.ид}}; \\ \tilde{k}_6 = 2,..,(k_3 + k_{10}); \\ k_{10} = 1,..,k_{\text{необр.рег.уч}}. \quad (5.9)$$

Седьмая величина представляет собой плотность распределения времени активной реакции на угрозу, обнаруженную и отраженную подсистемой обеспечения целостности данных:

$$f_7(\tilde{k}_7) = p_{int1} \cdot p_{react3} \cdot \sum_{k'} f_{обн.отр.цел.}(k') \cdot f_{акт.реакц.}(\tilde{k}_7 - k');$$

$$k' = k_4; k_4 = 1,.., k_{обн.отр.цел.}; k' = 1,.., k_{обн.отр.цел.};$$

$$\tilde{k}_7 = 2,..,(k_4 + k_{15});$$

$$k_{15} = 1,.., k_{акт.реакц.3}.$$
(5.10)

Восьмая величина представляет собой плотность распределения времени пассивной реакции на угрозу, обнаруженную и отраженную подсистемой обеспечения целостности данных:

$$f_8(\tilde{k}_8) = p_{int1} \cdot (1 - p_{react3}) \cdot \sum_{k'} f_{обн.отр.цел.}(k') \cdot f_{пасс.реакц.}(\tilde{k}_8 - k');$$

$$k' = k_4; k_4 = 1,.., k_{обн.отр.цел.}; k' = 1,.., k_{обн.отр.цел.};$$

$$\tilde{k}_8 = 2,..,(k_4 + k_{16});$$

$$k_{16} = 1,.., k_{пасс.реакц.3}.$$
(5.11)

Девятая величина представляет собой плотность распределения времени активной реакции на угрозу, обнаруженную и неотраженную подсистемой обеспечения целостности данных:

$$f_9(\tilde{k}_9) = p_{int2} \cdot p_{react4} \cdot \sum_{k'} f_{обн.неотр.цел.}(k') \cdot f_{акт.реакц.}(\tilde{k}_9 - k');$$

$$k' = k_5; k_5 = 1,.., k_{обн.неотр.цел.}; k' = 1,.., k_{обн.неотр.цел.};$$

$$\tilde{k}_9 = 2,..,(k_5 + k_{17});$$

$$k_{17} = 1,.., k_{акт.реакц.4}.$$
(5.12)

Десятая величина представляет собой плотность распределения времени пассивной реакции на угрозу, обнаруженную и неотраженную подсистемой обеспечения целостности данных:

$$f_{10}(\tilde{k}_{10}) = p_{int2} \cdot (1 - p_{react4}) \cdot \sum_{k'} f_{обн.неотр.цел.}(k') \cdot f_{пасс.реакц.}(\tilde{k}_{10} - k');$$

$$k' = k_5; k_5 = 1,.., k_{обн.неотр.цел.}; k' = 1,.., k_{обн.неотр.цел.};$$

$$\tilde{k}_{10} = 2,..,(k_5 + k_{18});$$

$$k_{18} = 1,.., k_{пасс.реакц.4}.$$
(5.13)

Одинарная величина представляет собой плотность распределения времени обнаружения и неотражения угрозы подсистемой обеспечения целостности данных:

$$f_{11}(\tilde{k}_{11}) = (1 - p_{int1} - p_{int2}) \cdot f_{\text{необн.неотр.цел.}}(k_6);$$

$$\tilde{k}_{11} = 1,..,k_6;$$

$$k_6 = 1,..,k_{\text{необн.неотр.цел.}}$$
(5.14)

После параллельного объединения ветвей графа по логическим функциям получаем следующие промежуточные результаты:

$$q_1(\hat{k}_1) = f_3(\hat{k}_1) \sum_{\tilde{k}_7 \leq \hat{k}_1} f_7(\tilde{k}_7) + f_7(\hat{k}_1) \sum_{\tilde{k}_3 < \hat{k}_1} f_3(\tilde{k}_3);$$

$$\hat{k}_1 = \max(\min(\tilde{k}_3), \min(\tilde{k}_7)), \dots, \max(\max(\tilde{k}_3), \max(\tilde{k}_7));$$

$$\tilde{k}_3 = 2,..,(k_2 + k_8);$$

$$k_2 = 1,..,k_{\text{необн..отр..ид.}}; k_8 = 1,..,k_{\text{необн..рег..уч.}};$$

$$\tilde{k}_7 = 2,..,(k_4 + k_{15});$$

$$k_4 = 1,..,k_{\text{обн..отр..цел.}}; k_{15} = 1,..,k_{\text{акт..реакц..3.}}$$
(5.15)

$$q_2(\hat{k}_2) = f_1(\hat{k}_2) \sum_{\tilde{k}_7 \leq \hat{k}_2} f_7(\tilde{k}_7) + f_7(\hat{k}_2) \sum_{\tilde{k}_1 < \hat{k}_2} f_1(\tilde{k}_1);$$

$$\hat{k}_2 = \max(\min(\tilde{k}_1), \min(\tilde{k}_7)), \dots, \max(\max(\tilde{k}_1), \max(\tilde{k}_7));$$

$$\tilde{k}_1 = \min(k_1 + k_{11}, k_2 + k_7 + k_{11}), \dots, \max(k_1 + k_{11}, k_2 + k_7 + k_{11});$$

$$k_1 = 1,..,k_{\text{обн..отр..ид.}}; k_2 = 1,..,k_{\text{необн..отр..ид.}};$$

$$k_7 = 1,..,k_{\text{обн..рег..уч.}}; k_{11} = 1,..,k_{\text{акт..реакц..1.}}$$

$$\tilde{k}_7 = 2,..,(k_4 + k_{15});$$

$$k_4 = 1,..,k_{\text{обн..отр..цел.}}; k_{15} = 1,..,k_{\text{акт..реакц..3.}}$$
(5.16)

$$\begin{aligned}
q_3(\hat{k}_3) &= f_1(\hat{k}_3) \sum_{\tilde{k}_8 \leq \hat{k}_3} f_8(\tilde{k}_8) + f_8(\hat{k}_3) \sum_{\tilde{k}_1 < \hat{k}_3} f_1(\tilde{k}_1); \\
\hat{k}_3 &= \max(\min(\tilde{k}_1), \min(\tilde{k}_8)), \dots, \max(\max(\tilde{k}_1), \max(\tilde{k}_8)); \\
\tilde{k}_1 &= \min(k_1 + k_{11}, k_2 + k_7 + k_{11}), \dots, \max(k_1 + k_{11}, k_2 + k_7 + k_{11}); \\
k_1 &= 1, \dots, k_{\text{обн.опр.и.д.}}; k_2 = 1, \dots, k_{\text{необн.опр.и.д.}}; \\
k_7 &= 1, \dots, k_{\text{обн.рег.уч.}}; k_{11} = 1, \dots, k_{\text{акт.peaky.1}}; \\
\tilde{k}_8 &= 2, \dots, (k_4 + k_{16}); \\
k_4 &= 1, \dots, k_{\text{обн.опр.и.ел.}}; k_{16} = 1, \dots, k_{\text{nacc.peaky.3}}. 
\end{aligned} \tag{5.17}$$

$$\begin{aligned}
q_4(\hat{k}_4) &= f_2(\hat{k}_4) \sum_{\tilde{k}_7 \leq \hat{k}_4} f_7(\tilde{k}_7) + f_7(\hat{k}_4) \sum_{\tilde{k}_2 < \hat{k}_4} f_2(\tilde{k}_2); \\
\hat{k}_4 &= \max(\min(\tilde{k}_2), \min(\tilde{k}_7)), \dots, \max(\max(\tilde{k}_2), \max(\tilde{k}_7)); \\
\tilde{k}_2 &= \min(k_1 + k_{12}, k_2 + k_7 + k_{12}), \dots, \max(k_1 + k_{12}, k_2 + k_7 + k_{12}); \\
k_1 &= 1, \dots, k_{\text{обн.опр.и.д.}}; k_2 = 1, \dots, k_{\text{необн.опр.и.д.}}; \\
k_7 &= 1, \dots, k_{\text{обн.рег.уч.}}; k_{12} = 1, \dots, k_{\text{nacc.peaky.1}}; \\
\tilde{k}_7 &= 2, \dots, (k_4 + k_{15}); \\
k_4 &= 1, \dots, k_{\text{обн.опр.и.ел.}}; k_{15} = 1, \dots, k_{\text{акт.peaky.3}}. 
\end{aligned} \tag{5.18}$$

$$\begin{aligned}
q_5(\hat{k}_5) &= f_2(\hat{k}_5) \sum_{\tilde{k}_8 \leq \hat{k}_5} f_8(\tilde{k}_8) + f_8(\hat{k}_5) \sum_{\tilde{k}_2 < \hat{k}_5} f_2(\tilde{k}_2); \\
\hat{k}_5 &= \max(\min(\tilde{k}_2), \min(\tilde{k}_8)), \dots, \max(\max(\tilde{k}_2), \max(\tilde{k}_8)); \\
\tilde{k}_2 &= \min(k_1 + k_{12}, k_2 + k_7 + k_{12}), \dots, \max(k_1 + k_{12}, k_2 + k_7 + k_{12}); \\
k_1 &= 1, \dots, k_{\text{обн.опр.и.д.}}; k_2 = 1, \dots, k_{\text{необн.опр.и.д.}}; \\
k_7 &= 1, \dots, k_{\text{обн.рег.уч.}}; k_{12} = 1, \dots, k_{\text{nacc.peaky.1}}; \\
\tilde{k}_8 &= 2, \dots, (k_4 + k_{16}); \\
k_4 &= 1, \dots, k_{\text{обн.опр.и.ел.}}; k_{16} = 1, \dots, k_{\text{nacc.peaky.3}}. 
\end{aligned} \tag{5.19}$$

$$\begin{aligned}
q_6(\hat{k}_6) &= f_3(\hat{k}_6) \sum_{\tilde{k}_8 \leq \hat{k}_6} f_8(\tilde{k}_8) + f_8(\hat{k}_6) \sum_{\tilde{k}_3 < \hat{k}_6} f_3(\tilde{k}_3); \\
\hat{k}_6 &= \max(\min(\tilde{k}_3), \min(\tilde{k}_8)), \dots, \max(\max(\tilde{k}_3), \max(\tilde{k}_8)); \\
\tilde{k}_3 &= 2, \dots, (k_2 + k_8); \\
k_2 &= 1, \dots, k_{\text{необн.опр.ио}}; k_8 = 1, \dots, k_{\text{необн.рег.уч.}}; \\
\tilde{k}_8 &= 2, \dots, (k_4 + k_{16}); \\
k_4 &= 1, \dots, k_{\text{обн.опр.чел.}}; k_{16} = 1, \dots, k_{\text{nacc.peaky.3}}.
\end{aligned} \tag{5.20}$$

$$\begin{aligned}
q_7(\hat{k}_7) &= f_4(\hat{k}_7)(1 - \sum_{\tilde{k}_9 \leq \hat{k}_7} f_9(\tilde{k}_9)) + f_9(\hat{k}_7)(1 - \sum_{\tilde{k}_4 < \hat{k}_7} f_4(\tilde{k}_4)); \\
\hat{k}_7 &= \min(\min(\tilde{k}_4), \min(\tilde{k}_9)), \dots, \min(\max(\tilde{k}_4), \max(\tilde{k}_9)); \\
\tilde{k}_4 &= 3, \dots, (k_3 + k_9 + k_{13}); \\
k_3 &= 1, \dots, k_{\text{необн.неопр.ио}}; k_9 = 1, \dots, k_{\text{обн.рег.уч.}}; k_{13} = 1, \dots, k_{\text{акт.peaky.2}}; \\
\tilde{k}_9 &= 2, \dots, (k_5 + k_{17}); \\
k_5 &= 1, \dots, k_{\text{обн.неопр.чел.}}; k_{17} = 1, \dots, k_{\text{акт.peaky.4}}.
\end{aligned} \tag{5.21}$$

$$\begin{aligned}
q_8(\hat{k}_8) &= f_5(\hat{k}_8) \sum_{\tilde{k}_{10} \leq \hat{k}_8} f_{10}(\tilde{k}_{10}) + f_{10}(\hat{k}_8) \sum_{\tilde{k}_5 < \hat{k}_8} f_5(\tilde{k}_5); \\
\hat{k}_8 &= \max(\min(\tilde{k}_5), \min(\tilde{k}_{10})), \dots, \max(\max(\tilde{k}_5), \max(\tilde{k}_{10})); \\
\tilde{k}_5 &= 3, \dots, (k_3 + k_9 + k_{14}); \\
k_3 &= 1, \dots, k_{\text{необн.неопр.ио}}; k_9 = 1, \dots, k_{\text{обн.рег.уч.}}; k_{14} = 1, \dots, k_{\text{nacc.peaky.2}}; \\
\tilde{k}_{10} &= 2, \dots, (k_5 + k_{18}); \\
k_5 &= 1, \dots, k_{\text{обн.неопр.чел.}}; k_{18} = 1, \dots, k_{\text{nacc.peaky.4}}.
\end{aligned} \tag{5.22}$$

$$\begin{aligned}
q_9(\hat{k}_9) &= f_3(\hat{k}_9) \sum_{\tilde{k}_{11} \leq \hat{k}_9} f_{11}(\tilde{k}_{11}) + f_{11}(\hat{k}_9) \sum_{\tilde{k}_3 < \hat{k}_9} f_3(\tilde{k}_3); \\
\hat{k}_9 &= \max(\min(\tilde{k}_3), \min(\tilde{k}_{11})), \dots, \max(\max(\tilde{k}_3), \max(\tilde{k}_{11})); \\
\tilde{k}_3 &= 2, \dots, (k_2 + k_8); \\
k_2 &= 1, \dots, k_{\text{необн.опр.ио}}; k_8 = 1, \dots, k_{\text{необн.рег.уч.}}; \\
\tilde{k}_{11} &= 1, \dots, k_6; \\
k_6 &= 1, \dots, k_{\text{необн.неопр.чел.}}
\end{aligned} \tag{5.23}$$

$$\begin{aligned}
q_{10}(\hat{k}_{10}) &= f_6(\hat{k}_{10}) \sum_{k_1 \leq \hat{k}_{10}} f_{11}(\tilde{k}_1) + f_{11}(\hat{k}_{10}) \sum_{\tilde{k}_6 < \hat{k}_{10}} f_6(\tilde{k}_6); \\
\hat{k}_{10} &= \max(\min(\tilde{k}_6), \min(\tilde{k}_{11})), \dots, \max(\max(\tilde{k}_6), \max(\tilde{k}_{11})); \\
\tilde{k}_6 &= 2, \dots, (k_3 + k_{10}); \\
k_3 &= 1, \dots, k_{\text{необн.неотр.ид.}}; k_{10} = 1, \dots, k_{\text{необн.рег.уч.}}; \\
\tilde{k}_{11} &= 1, \dots, k_6; \\
k_6 &= 1, \dots, k_{\text{необн.неотр.чел.}}
\end{aligned} \tag{5.24}$$

Вывод окончательных выражений для плотностей распределения времени принятия итоговых решений:

$$\begin{aligned}
f_{\text{акт.обн.и.отр.}}(\dot{k}_1) &= \sum_{\hat{k}_1} q_1(\hat{k}_1) \cdot f_{\text{зад.реш.0}}(\dot{k}_1 - \hat{k}_1) + \sum_{\hat{k}_6} q_2(\hat{k}_6) \cdot f_{\text{зад.реш.1}}(\dot{k}_1 - \hat{k}_6) + \\
&+ \sum_{\hat{k}_1} q_3(\hat{k}_3) \cdot f_{\text{зад.реш.2}}(\dot{k}_1 - \hat{k}_3) + \sum_{\hat{k}_6} q_4(\hat{k}_4) \cdot f_{\text{зад.реш.3}}(\dot{k}_1 - \hat{k}_4); \\
\dot{k}_1 &= \min(\hat{k}_1 + k_{19}, \hat{k}_2 + k_{20}, \hat{k}_3 + k_{21}, \hat{k}_4 + k_{22}), \dots, \\
&\dots, \max(\hat{k}_1 + k_{19}, \hat{k}_2 + k_{20}, \hat{k}_3 + k_{21}, \hat{k}_4 + k_{22});
\end{aligned}$$

$$\begin{aligned}
\hat{k}_1 &= \max(\min(\tilde{k}_3), \min(\tilde{k}_7)), \dots, \max(\max(\tilde{k}_3), \max(\tilde{k}_7)); \\
\tilde{k}_3 &= 2, \dots, (k_2 + k_8); \\
k_2 &= 1, \dots, k_{\text{необн.отр.ид.}}; k_8 = 1, \dots, k_{\text{необн.рег.уч.}}; \\
\tilde{k}_7 &= 2, \dots, (k_4 + k_{15}); \\
k_4 &= 1, \dots, k_{\text{обн.отр.чел.}}; k_{15} = 1, \dots, k_{\text{акт.реаку.3}};
\end{aligned}$$

$$\begin{aligned}
\hat{k}_3 &= \max(\min(\tilde{k}_1), \min(\tilde{k}_8)), \dots, \max(\max(\tilde{k}_1), \max(\tilde{k}_8)); \\
\tilde{k}_1 &= \min(k_1 + k_{11}, k_2 + k_7 + k_{11}), \dots, \max(k_1 + k_{11}, k_2 + k_7 + k_{11}); \\
k_1 &= 1, \dots, k_{\text{обн.отр.ид.}}; k_2 = 1, \dots, k_{\text{необн.отр.ид.}}; \\
k_7 &= 1, \dots, k_{\text{обн.рег.уч.}}; k_{11} = 1, \dots, k_{\text{акт.реаку.1}}; \\
\tilde{k}_8 &= 2, \dots, (k_4 + k_{16}); \\
k_4 &= 1, \dots, k_{\text{обн.отр.чел.}}; k_{16} = 1, \dots, k_{\text{нacc.реаку.3}};
\end{aligned}$$

$$\begin{aligned}
\hat{k}_4 &= \max(\min(\tilde{k}_2), \min(\tilde{k}_7)), \dots, \max(\max(\tilde{k}_2), \max(\tilde{k}_7)); \\
\tilde{k}_2 &= \min(k_1 + k_{12}, k_2 + k_7 + k_{12}), \dots, \max(k_1 + k_{12}, k_2 + k_7 + k_{12}); \\
k_1 &= 1, \dots, k_{\text{обн.опр.и.о.}}; k_2 = 1, \dots, k_{\text{необн.опр.и.о.}}; \\
k_7 &= 1, \dots, k_{\text{обн.рез.у.ч.}}; \tilde{k}_{12} = 1, \dots, k_{\text{насс.peaky..1}}; \\
\tilde{k}_7 &= 2, \dots, (k_4 + k_{15}); \\
k_4 &= 1, \dots, k_{\text{обн.опр.чел.}}; k_{15} = 1, \dots, k_{\text{акт.peaky..3}}.
\end{aligned} \tag{5.25}$$

$$\begin{aligned}
f_{\text{насс.обн.и.опр.}}(\hat{k}_2) &= \sum_{\hat{k}_5} q_5(\hat{k}_5) \cdot f_{\text{зад.реш.4}}(\hat{k}_2 - \hat{k}_5) + \sum_{\hat{k}_6} q_6(\hat{k}_6) \cdot f_{\text{зад.реш.5}}(\hat{k}_2 - \hat{k}_6); \\
\hat{k}_2 &= \min(\hat{k}_5 + k_{23}, \hat{k}_6 + k_{24}), \dots, \max(\hat{k}_5 + k_{23}, \hat{k}_6 + k_{24}); \\
\hat{k}_5 &= \max(\min(\tilde{k}_2), \min(\tilde{k}_8)), \dots, \max(\max(\tilde{k}_2), \max(\tilde{k}_8)); \\
\tilde{k}_2 &= \min(k_1 + k_{12}, k_2 + k_7 + k_{12}), \dots, \max(k_1 + k_{12}, k_2 + k_7 + k_{12}); \\
k_1 &= 1, \dots, k_{\text{обн.опр.и.о.}}; k_2 = 1, \dots, k_{\text{необн.опр.и.о.}}; \\
k_7 &= 1, \dots, k_{\text{обн.рез.у.ч.}}; k_{12} = 1, \dots, k_{\text{насс.peaky..1}}; \\
\tilde{k}_8 &= 2, \dots, (k_4 + k_{16}); \\
k_4 &= 1, \dots, k_{\text{обн.опр.чел.}}; k_{16} = 1, \dots, k_{\text{насс.peaky..3}}; \\
\hat{k}_6 &= \max(\min(\tilde{k}_3), \min(\tilde{k}_8)), \dots, \max(\max(\tilde{k}_3), \max(\tilde{k}_8)); \\
\tilde{k}_3 &= 2, \dots, (k_2 + k_8); \\
k_2 &= 1, \dots, k_{\text{необн.опр.и.о.}}; k_8 = 1, \dots, k_{\text{необн.рез.у.ч.}}; \\
\tilde{k}_8 &= 2, \dots, (k_4 + k_{16}); \\
k_4 &= 1, \dots, k_{\text{обн.опр.чел.}}; k_{16} = 1, \dots, k_{\text{насс.peaky..3}}.
\end{aligned} \tag{5.26}$$

$$\begin{aligned}
f_{\text{акт.обн.и.неопр.}}(\hat{k}_3) &= \sum_{\hat{k}_7} q_7(\hat{k}_7) \cdot f_{\text{зад.реш.6}}(\hat{k}_3 - \hat{k}_7); \\
\hat{k}_3 &= \min(\hat{k}_7 + k_{25}), \dots, \max(\hat{k}_7 + k_{25}); \\
\hat{k}_7 &= \min(\min(\tilde{k}_4), \min(\tilde{k}_9)), \dots, \min(\max(\tilde{k}_4), \max(\tilde{k}_9)); \\
\tilde{k}_4 &= 3, \dots, (k_3 + k_9 + k_{13}); \\
k_3 &= 1, \dots, k_{\text{необн.неопр.и.о.}}; k_9 = 1, \dots, k_{\text{обн.рез.у.ч.}}; k_{13} = 1, \dots, k_{\text{акт.peaky..2}}; \\
\tilde{k}_9 &= 2, \dots, (k_5 + k_{17}); \\
k_5 &= 1, \dots, k_{\text{обн.неопр.чел.}}; k_{17} = 1, \dots, k_{\text{акт.peaky..4}}.
\end{aligned} \tag{5.27}$$

$$\begin{aligned}
f_{\text{насс.обн.и.неотр.}}(\dot{k}_4) &= \sum_{\hat{k}_8} q_8(\hat{k}_8) \cdot f_{\text{зад.реш.7}}(\dot{k}_4 - \hat{k}_8); \\
\dot{k}_4 &= \min(\hat{k}_8 + k_{26}), \dots, \max(\hat{k}_8 + k_{26}); \\
\hat{k}_8 &= \max(\min(\tilde{k}_5), \min(\tilde{k}_{10})), \dots, \max(\max(\tilde{k}_5), \max(\tilde{k}_{10})); \\
\tilde{k}_5 &= 3, \dots, (k_3 + k_9 + k_{14}); \\
k_3 &= 1, \dots, k_{\text{необн.неотр.ид.}}; k_9 = 1, \dots, k_{\text{обн.рег.уч.}}; k_{14} = 1, \dots, k_{\text{насс.реакц.2}}; \\
\tilde{k}_{10} &= 2, \dots, (k_5 + k_{18}); \\
k_5 &= 1, \dots, k_{\text{обн.неотр.чел.}}; k_{18} = 1, \dots, k_{\text{насс.реакц.4}}.
\end{aligned} \tag{5.28}$$

$$\begin{aligned}
f_{\text{необн.неотр.}}(\dot{k}_5) &= \sum_{\hat{k}_9} q_9(\hat{k}_9) \cdot f_{\text{зад.реш.8}}(\dot{k}_5 - \hat{k}_9) + \sum_{\hat{k}_{10}} q_{10}(\hat{k}_{10}) \cdot f_{\text{зад.реш.9}}(\dot{k}_5 - \hat{k}_{10}); \\
\dot{k}_5 &= \min(\hat{k}_9 + k_{27}, \hat{k}_{10} + k_{28}), \dots, \max(\hat{k}_9 + k_{27}, \hat{k}_{10} + k_{28}); \\
\hat{k}_9 &= \max(\min(\tilde{k}_3), \min(\tilde{k}_{11})), \dots, \max(\max(\tilde{k}_3), \max(\tilde{k}_{11})); \\
\tilde{k}_3 &= 2, \dots, (k_2 + k_8); \\
k_2 &= 1, \dots, k_{\text{необн.отр.ид.}}; k_8 = 1, \dots, k_{\text{необн.рег.уч.}}; \\
\tilde{k}_{11} &= 1, \dots, k_6; \\
k_6 &= 1, \dots, k_{\text{необн.неотр.чел.}}; \\
\hat{k}_{10} &= \max(\min(\tilde{k}_6), \min(\tilde{k}_{11})), \dots, \max(\max(\tilde{k}_6), \max(\tilde{k}_{11})); \\
\tilde{k}_6 &= 2, \dots, (k_3 + k_{10}); \\
k_3 &= 1, \dots, k_{\text{необн.неотр.ид.}}; k_{10} = 1, \dots, k_{\text{необн.рег.уч.}}; \\
\tilde{k}_{11} &= 1, \dots, k_6; \\
k_6 &= 1, \dots, k_{\text{необн.неотр.чел.}}
\end{aligned} \tag{5.29}$$

Из полученных соотношений находятся числовые характеристики искомого времени. Математическое ожидание и дисперсия времени принятия соответствующего решения определяются по соотношениям:

$$\begin{aligned}
M[k] &= \sum_{k=1}^K k \cdot f_T(k); \\
D[k] &= \sum_{k=1}^K (k - M\{k\})^2 \cdot f_T(k).
\end{aligned} \tag{5.30}$$

Моменты более высокого порядка получаются следующим образом:

$$M_n[k] = \sum_{k=1}^K (k - M[k])^n \cdot f_T(k). \quad (5.31)$$

Реализацию данных вычислений с использованием библиотеки MPI можно осуществить посредством распараллеливания операций, требующих обработки большого количества данных.

### 5.3. Описание программы

Программная документация приводится в Приложении 3.

#### 5.3.1. Работа с программой

Для работы программы необходимо чтобы на каждом узле, где будет запускаться приложение, был установлен пакет MPICH и скопирован файл Parallel.exe. Для работы с приложением может использоваться командная строка или графический интерфейс.

Командная строка должна иметь формат вызова в соответствии с требованиями для запуска программ с помощью загрузчика mpirun.

Пример вызова программы из командной строки:

```
mpirun -np 20 -localonly parallel.exe
```

При работе с визуальным интерфейсом необходимо ввести все требуемые данные: вероятности, времена выполнения подпроцессов и закон распределения. Общий вид графического интерфейса приведен на рис 5.6.

При запуске программы на базе всех введенных данных формируется конфигурационный файл, с указанием имени которого происходит системный вызов mpirun.

Пример конфигурационного файла, который создается автоматически, приведен ниже.

```
exe parallel.exe
args 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
0.2 0.4 0.4 0.1 0.2 0.5 0.5 0.2 0.7 0.3 0
hosts
termit 2
```

Общая структура указанного файла следующая:

```
exe parallel.exe
```

```

args k1 k2 k3... k28 pid1 pid2 preg1 preg2 pint1 pint2 preact1
preact2 preact3 preact4 Закон_распределения
hosts
Имя_удаленного_компьютера_1 Количество_процессов
Имя_удаленного_компьютера_2 Количество_процессов

```

Для загрузки времен выполнения подпроцессов надо нажать кнопку **Load times**, в результате чего начнется стандартный диалог, в котором требуется выбрать файл, содержащий эти данные, или заполнить соответствующие поля на форме. Загрузка вероятностей происходит аналогично, но при нажатии кнопки **Load probability**.

Для указания компьютеров, на которых будет запускаться приложение, необходимо добавить имя удаленной машины и количество процессов, заполнив соответствующие поля и нажав кнопку **Add**. Все вводимые значения должны быть корректными иначе выдается предупреждение.

При определении исходных данных необходимо выбрать закон распределения в подпроцессах, по умолчанию устанавливается экспоненциальный закон.

Для выполнения программы с использованием MPI необходимо нажать кнопку **Начать вычисление**.

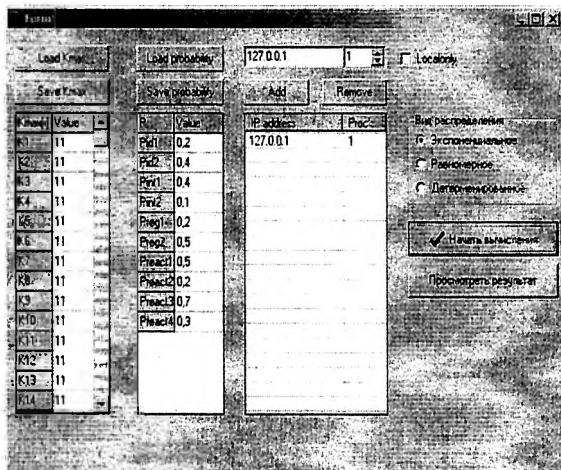


Рис. 5.6. Общий вид графического интерфейса

После вычислений можно просмотреть полученные результаты в графическом виде, для этого необходимо нажать кнопку

Просмотреть результат

После ее нажатия появятся графики, отображающие функции плотностей распределения времени выполнения подпроцессов для каждого из пяти итоговых решений. Для каждой функции создается отдельный график. Пример получаемого графика для экспоненциального распределения приведен на рис. П3.2.

### 5.3.2. Результаты экспериментов

Данные, полученные в результате экспериментов с использованием библиотеки MPI и пакета MathCad, идентичны. В табл. П3.2, П3.3, П3.4, П3.5 приводятся их значения.

Корректность используемых методов расчета статистических характеристик подтвердим, задав в качестве исходных данных для этих вычислений последовательность чисел с фиксированным законом распределения, генерируемую пакетом MathCad.

Вычисление математического ожидания и моментов более высоких порядков осуществляется следующими процедурами:

```
MO(f) := | row ← 1
           Mo ← 0
           for i ∈ 1..rows(f)
               Mo ← i fi + Mo
           Mo
```

```
Mn(p,f) := | m ← 0
               for i ∈ 1..rows(f)
                   m ← (fi - MO(f))p · fi + m
               m
```

Асимметрия и эксцесс вычисляются по уже известным соотношениям:

$$b_1 := \frac{M(3, MOg(y), y)^2}{M(2, MOg(y), y)^3}$$

$$b_2 := \frac{M(4, MOg(y), y)^2}{M(2, MOg(y), y)^3}$$

Генерация чисел, распределенных по экспоненциальному закону, производится функцией `rexp()`, по равномерному закону – функцией `runif()`.

В результате вычислений при размере генерируемого вектора 500 элементов получаем следующие значения асимметрии и эксцесса. Для экспоненциального закона распределения асимметрия равна 3,8, эксцесс 8,56. Для равномерного закона распределения асимметрия равна 0, эксцесс 1,26.

Полученные значения лежат близко к табличным, что свидетельствует о корректности использования такого рода вычислений в созданном программном обеспечении.

О правильности получаемых распределений на выходе можно судить и по внешнему виду получаемых функций. На рис. П3.2 показан график, полученный в результате обработки данных визуальным интерфейсом, в процессе эксперимента 1, и на рис. П3.3 график, полученный в пакете MathCad.

Временные затраты на выполнение программы в зависимости от размера входных данных показаны на рис.П3.4. Под размерностью входных данных подразумевается максимальное время работы подпроцессов, одинаковое для всех экземпляров.

Как видно из рис.П3.4, использование параллельности при малых размерностях не дает преимуществ. Затраты на коммуникационные обмены оказываются больше, чем время выполнения вычислений, однако с некоторого момента наблюдается следующая тенденция: чем больше реальных процессов задействовано, тем меньше время выполнения программы.

#### 5.4. Выводы

1. Построенная базовая модель процессов обнаружения и отражения угроз для автоматизированных систем третьего класса защищенности расширяет пространство моделирования систем защиты информации.
2. Выведенные аналитические соотношения для определения динамических характеристик автоматизированных систем третьего класса защищенности отражают значимость эффекта распараллеливания при достижении оперативности систем защиты информации.
3. Разработанный прототип параллельной программы с функциями библиотеки MPI выполняет задачу оценки статистических

характеристик времени обнаружения и отражения угроз в автоматизированных системах третьего класса защищённости с последовательно-параллельной и распределенной обработкой информации.

4. Согласование результатов выполнения созданного программного средства с данными, полученными с применением пакета MathCad, подтверждает работоспособность и высокую точность результатов построенного прототипа параллельной программы с функциями MPI-библиотеки для оценки статистических характеристик времени обнаружения и отражения угроз в автоматизированных системах третьего класса защищённости с последовательно-параллельной и распределенной обработкой информации.
5. Построенный . прототип программы в базисе функций MPI предназначен для оценки динамических характеристик параллельных вычислений.

## **ЗАКЛЮЧЕНИЕ**

В учебном пособии обоснована актуальность развития технологических приемов интеллектуализации программного обеспечения высокопроизводительных вычислительных систем. Раскрыты технологические приемы нацелены на преодоление отчуждения процессов мониторинга качества функционирования параллельных программ и распределенных приложений от процессов реализации поставленных пред ними задач.

В основных разделах пособия представлена научно-обоснованная система прототипов ключевых программных компонентов в базисе функций библиотеки MPI для определения качества работы программного обеспечения высокопроизводительных вычислительных систем. В процессе представления указанной системы решены следующие задачи:

1. Определены характеристики спецификаций базиса функций библиотеки MPI.
2. Разработаны прототипы базовых компонентов программного обеспечения в базисе функций библиотеки MPI для оценки статистических характеристик времени выполнения параллельных программ.
3. Выполнен анализ работоспособности базовых компонентов программного обеспечения и подтверждена корректность их функционирования.
4. Построена базовая модель процессов обнаружения и отражения угроз для автоматизированных систем третьего класса защищенности с последовательно-параллельной и распределенной обработкой информации.
5. Выведены аналитические соотношения для определения динамических характеристик автоматизированных систем третьего класса защищенности с последовательно-параллельной и распределенной обработкой информации.
6. Разработан и описан прототип программы для оценки статистических характеристик обнаружения и отражения угроз для автоматизированных систем третьего класса защищенности в базисе функций MPI.
7. Создан прототип программы в базисе функций MPI для определения динамических характеристик параллельных вычислений.

При решении перечисленных задач отражены следующие результаты:

- 1) Выявлены общие тенденции развития современных подходов к разработке и исследованию программного обеспечения для высокопроизводительных вычислительных систем.
- 2) Определена позиция библиотеки MPI в составе средств программирования параллельных вычислений.
- 3) Составлены основные функциональные группы функций библиотеки MPI и раскрыты их ключевые особенности.
- 4) Разработаны различные варианты базовых компонентов в базисе функций MPI при объединении подпрограмм по логическим функциям «И», «ИЛИ», «И-ИЛИ».
- 5) Выбраны лучшие варианты базовых компонентов.
- 6) Подтверждена корректность выполняемых вычислений.
- 7) Выведены аналитические соотношения для модели, описывающей функционирование системы безопасности третьего класса защищенности.
- 8) Разработано математическое обеспечение для определения динамических характеристик систем защиты информации третьего класса защищенности.
- 9) Подтверждена корректная работа созданного программного обеспечения и проведено его тестирование

Научную новизну представляют следующие результаты:

1. Предложен новый подход к формированию шаблонов для программного обеспечения высокопроизводительных вычислительных систем, способного динамически регулировать качество выполняемых задач в среде MPI.
2. Расширен состав моделей, описывающих функционирование систем третьего класса защищенности с позиций информационной безопасности.
3. Развито математическое обеспечение для определения динамических характеристик систем защиты информации, соответствующих третьему классу защищенности в условиях последовательно-параллельной и распределенной обработки информации.

Практическую значимость имеют следующие результаты:

- 1) Расширен состав информационного и математического обеспечения образовательных программ подготовки бакалавров и магистров по направлениям «Информационные технологии», «Прикладная математика и информатика», «Системный анализ и управление»,

- «Информатика и вычислительная техника», «Информационные системы».
- 2) Создан набор программных прототипов для создания интеллектуального программного обеспечения.
  - 3) Сформировано множество альтернативных способов организации шаблонов, учитывающих ключевые особенности типовых топологий параллельных вычислений, применяемых на практике.
  - 4) Построены зависимости времени выполнения программных прототипов от размерностей входных данных, позволяющие выбрать их рациональное сочетание.

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. – М.: Мир, 1982. – 416 с.
2. Андрианов А.Н., Ефимкин К.Н., Задыхайло И.Б. Язык Норма. Препринт ИПМ им. М.В. Келдыша АН СССР, № 165, 1985.
3. Задыхайло И.Б., Крюков В.А., Поздняков Л.А. RAMPA – CASE для разработки мобильных параллельных программ, протокол международной конференции параллельных вычислительных технологий, Обнинск, 1993.
4. Коновалов Н.А., Крюков В.А., Михайлов С.Н., Погребцов А. А. Fortran DVM – язык разработки мобильных параллельных программ. Принципы работы программного обеспечения мультипроцессорных и суперкомпьютерных систем: Теория, Практика, Опыт. Институт Системного программирования, Москва, 1994.
5. Lastovetsky A. mpC – a Multi-Paradigm Programming Language for Massively Parallel Computers, ACM SIGPLAN Notices, 31(2):13–20, February 1996.
6. Абрамов С., Адамович А., Коваленко М. Т-система: программное окружение, обеспечивающее автоматический динамический параллелизм в IP-сетях под управлением ОС Unix. Доклад четвертой международной российско-индийской выставки-конференции, Сент. 15-25, Москва, 1997.  
<http://www.botik.ru/~abram/ts-pubs.html>
7. Message-Passing Interface Forum, Document for a Standard Message Passing Interface, 1993. Version 1.0. <http://www.unix.mcs.anl.gov/mpi/>
8. Message-Passing Interface Forum, MPI-2: Extensions to the Message-Passing Interface, 1997. <http://www.unix.mcs.anl.gov/mpi/>
9. High Performance Fortran Forum. High Performance Fortran Language Specification. Version 1.0, May 1993.
10. High Performance Fortran Forum. High Performance Fortran Language Specification. Version 2.0, January 1997.
11. OpenMP Consortium: OpenMP Fortran Application Program Interface, Version 1.0, October 1997. <http://www.openmp.org/>
12. DVM-система. <http://www.keldysh.ru/dvm/>
13. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. – 600 с.
14. Букатов А.А., Дацюк В.Н., Жегуло А.И. Программирование многопроцессорных вычислительных систем. Ростов-на-Дону. ООО «ЦВВР», 2003. – 208 с.
15. Крюков В.А. Разработка параллельных программ для вычислительных кластеров и сетей. <http://www.keldysh.ru/>
16. PCF Fortran. Version 3.1. Aug.1, 1990.
17. Bailey D., Harris T., Saphir W., Van der Wijngaart, Woo A., Yarrow M. The NAS Parallel Benchmarks 2.0. NASA Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 1995. <http://science.nas.nasa.gov/Software/NPB>
18. TotalView. <http://www.etcus.com/Products/TotalView/index.html>
19. Portland Group Debugger. <http://www.pgroup.com>
20. Nupshot. <http://www.mcs.anl.gov/mpi/mpich/>
21. Pablo. <http://www-pablo.cs.uiuc.edu>
22. Vampir. <http://www.dallas.de/pages/vampir.htm>

23. Frumkin M., Jin H., and Yan J. Implementation of NAS Parallel Benchmarks in High Performance Fortran. NAS Technical Report NAS-98-009, NASA Ames Research Center, Moffett Field, CA, 1998.
24. Capello F., Etiemble D. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In Proceedings of Supercomputing '2000, 2000.
25. Елизаров Г.С., Забродин А.В., Левин В.К., Каратанов В.В., Корнеев В.В., Савин Г.И., Шабанов Б.М. Структура многопроцессорной вычислительной системы МВС-1000М // Труды Всероссийской научной конференции "Высокопроизводительные вычисления и их приложения", 30 октября – 2 ноября, Черноголовка, 2000.
26. Dongarra J., Walker D., and others. ScaLAPACK Users' Guide. Philadelphia: Society for Industrial and Applied Mathematics, 1997.
27. Корнеев В. Д. Параллельное программирование в MPI. Новосибирск: Из-во СО РАН, 2000. – 213 с.
28. Антонов А.С. Введение в параллельные вычисления (методическое пособие). М.: Из-во МГУ, 2002. – 70 с.
29. Мпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI. Пособие. Мн.: Из-во БГУ, 2002. – 323 с.
30. Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. СПб.: БХВ-Петербург, 2002. – 400 с.
31. Антонов А.С. Параллельное программирование с использованием технологии MPI. М.: Из-во МГУ, 2004.
32. Птицын А.В. Модель и метод анализа динамических характеристик систем защиты информации при автоматизированной обработке данных. Диссертация на соискание ученой степени кандидата технических наук. СПбГТУ, 1998. – 155 с.
33. Птицын А.В., Осовецкий Л.Г. Оценка динамических характеристик систем защиты информации //Современные научные школы; Перспективы развития (часть II). СПб.: СПбГТУ, 1998. – С.157–158.
34. Птицын А.В. Концепция создания комплексной СЗИНД для корпоративных сетей на базе СПДОП // Обеспечение безопасности и защиты информации сетевых технологий. Материалы семинара. СПб.: СПбГИТМО, 1999. – С.26–28.
35. Ian Foster. Designing and Building Parallel Programs. – <http://www.hensa.ac.uk/parallel/books/addison-wesley/dbpp>  
<http://rsusul1.rnd.runnet.ru/ncube/design/dbpp/book-info.html>
36. Сравнение реализаций MPI – <http://www.lfbs.rwth-aachen.de/>
37. Система тестов производительности для параллельных компьютеров – <http://parallel.ru/ftp/tests/>
38. Гостехкомиссия РФ. Руководящий документ. Автоматизированные системы. Защита от несанкционированного доступа к информации. Классификация автоматизированных систем и требования по защите информации. М.: Воениздат, 1992.
39. ГОСТ Р 51624–2000. Защита информации. Автоматизированные системы в защищенном исполнении. Общие требования.
40. ГОСТ Р 50992–96. Защита информации. Основные термины и определения.
41. ГОСТ Р 51275–99. Защита информации. Объект информатизации. Факторы, воздействующие на информацию. Общие положения.

42. ГОСТ Р 51583–2000. Защита информации. Порядок создания автоматизированных систем в защищенном исполнении. Общие положения.
43. Птицына Л.К., Дорофеева Е.В. Определение и оценка динамических характеристик автоматизированных систем третьего класса защищенности в базисе функций MPI // Материалы X Всероссийской конференции по проблемам науки и высшей школы «Фундаментальные исследования в технических университетах», СПб.: Издательство Политехнического университета, 2006. – С. 199 – 200.
44. Птицына Л.К., Птицын А.В. Развитие научно-методического обеспечения по защите информации // Материалы X Всероссийской конференции по проблемам науки и высшей школы «Фундаментальные исследования в технических университетах», СПб.: Издательство Политехнического университета, 2006. – С. 200 – 201.
45. Птицына Л.К., Воронов А.Д. Анализ динамических характеристик программного обеспечения комплексных систем защиты информации // Материалы X Всероссийской конференции по проблемам науки и высшей школы «Фундаментальные исследования в технических университетах», СПб.: Издательство Политехнического университета, 2006. – С. 202 – 203.
46. Птицына Л.К. Развитие научно-методического обеспечения по защите информации для образовательных программ высшего профессионального образования // Материалы межвузовской научно-методической конференции «Инновационные и научкоемкие технологии в высшем образовании России», М.: Издательство МИРЭА, 2006. – С. 43 – 46.
47. Птицына Л.К., Дорофеева Е.В. Оценка статистических характеристик обнаружения и отражения угроз для автоматизированных систем третьего класса защищенности в базисе функций MPI // Труды X Международной научно-практической конференции «Системный анализ в проектировании и управлении», СПб.: Издательство Политехнического университета, 2006. – С. 266 – 267.

## Описание стандарта MPI 1.1

### 1. Введение в MPI

Интерфейс Передачи сообщений (MPI) – считается очень успешным для написания параллельных программ. Реализация MPI существует для большинства параллельных вычислительных машин.

Причины успеха MPI не очевидны. Много пользователей и людей, изучающих MPI, сталкиваются с трудностями использования MPI. Обычно основные проблемы включают сложность MPI: большое число функций, отсутствие справки для компиляции и запуска, интеграция с основными языками программирования для упрощения работы с массивами, структурами и основными типами данных, а также сложность отладки программ. Много проблем сопутствующих возможности блокировки связей внутри программы. Тем не менее, MPI пользуется успехом у разработчиков программ.

Главное преимущество создания стандарта передачи сообщений состоит в его мобильности (portability) и простоте использования. В коммуникационной среде с распределенной памятью, в которой высший уровень процедур и/или абстракций построен над слоем процедур передачи сообщений, выгода стандартизации особенно очевидна. Более того, определение стандарта обеспечивает производителей четко определенным набором процедур, которые они могут эффективно реализовать или в некоторых случаях обеспечить аппаратную поддержку для них, увеличивая тем самым масштабируемость.

Целью MPI является создание широко используемого стандарта для написания программ на основе передачи сообщений. Следовательно, интерфейс должен быть практическим, мобильным, эффективным и гибким стандартом для передачи сообщений.

Полный список целей, приводимых в стандарте таков:

- разработать прикладной программный интерфейс (не обязательно для компиляторов или библиотеки системной реализации);
- обеспечить эффективный обмен, который позволял бы избегать копирования память-память, допускал совмещение операций обмена с вычислениями и разгружал коммуникационный сопроцессор, где это возможно;

- обеспечить удобные для интерфейса описания аргументов для языков С и ФОРТРАН77;
- обеспечить надежный коммуникационный интерфейс: с неисправностями должен справляться не пользователь, а нижележащие слои коммуникационной подсистемы;
- определить интерфейс, который не слишком отличается от уже существующих, таких как PVM, NX, Express, p4, и обеспечить расширения, которые допускают большую гибкость;
- определить интерфейс, который можно реализовать на большинстве платформ поставщиков без существенных изменений в нижележащей коммуникации и системном программном обеспечении;
- семантика интерфейса не должна зависеть от используемого языка программирования.

Интерфейс должен быть спроектирован так, чтобы позволять безопасное выполнение потоков

Стандарт описывает:

- парные обмены (point-to-point communication);
- коллективные операции (collective operations);
- группы процессов (process groups);
- коммуникационные контексты (communication contexts);
- привязки для языков ФОРТРАН77 и С;
- управление вычислительной средой и анализ (environmental management and inquiry);
- интерфейс профилирования (profiling interface).

Стандарт не описывает:

- явные операции с разделяемой памятью;
- операций, которые требуют большей поддержки операционной системы, чем есть в существующем стандарте; например, прием с прерыванием, удаленное исполнение или активные сообщения;
- пакеты для конструирования программ;
- возможности отладки;
- явную поддержку потоков;
- поддержку для управления задачами;
- функции ввода/вывода.

Имеется много особенностей, которые были рассмотрены, но не включены в этот стандарт. Это произошло по ряду причин, основная из них – ограниченное время для завершения стандарта. Следующая версия MPI частично расширяет предоставляемые возможности.

## *2. Термины, соглашения, основные понятия MPI*

Процедуры MPI описываются с помощью независимых от языка обозначений. Аргументы процедурных вызовов маркируются через IN, OUT или INOUT. Это означает, что:

- вызов использует, но не изменяет аргумент (IN);
- вызов может изменять аргумент (OUT);
- вызов использует и изменяет аргумент (INOUT).

Имеется один специальный случай: если аргумент является дескриптором скрытого объекта и объект модифицируется процедурой, тогда аргумент маркируется как OUT. Он маркируется таким образом, даже если сам дескриптор не модифицируется. Атрибут OUT используется для того, чтобы указать на модификацию дескрипторной ссылки.

Из-за повышенной возможности ошибок в MPI избегают использования аргумента INOUT для самых больших экстентов, особенно для скалярных аргументов.

Общим явлением для функций MPI является аргумент, который используется как IN некоторыми процессами и как OUT другими процессами. Такой аргумент маркируется как INOUT аргумент, хотя семантически он не используется ни в одном вызове одновременно как входной и выходной.

Часто возникает другая ситуация, когда значение аргумента необходимо только части процессов. Когда аргумент не является существенным для процесса, тогда в качестве аргумента может быть передано произвольное значение.

В стандарте MPI используются следующие семантические термины. Первые два обычно применяются для коммуникационных операций.

Неблокирующая операция – процедура возвращает управление перед завершением операции и перед тем, как пользователю разрешено повторно использовать ресурсы, указанные в вызове (такие, как буфер).

Блокирующая операция – возврат из процедуры указывает на то, что пользователю разрешено повторно использовать указанные в вызове ресурсы.

Локальная операция – завершение процедуры зависит только от локально выполняемого процесса. Такая операция не требует коммуникации с другим пользовательским процессом.

**Нелокальная операция** – завершение операции может потребовать выполнения некоторых процедур на другом процессе. Возможны коммуникации с другим пользовательским процессом.

**Коллективная операция** – все процессы в группе должны использовать эту процедуру.

Стандарт MPI фиксирует интерфейс, который должен соблюдаться как системой программирования на каждой вычислительной платформе, так и пользователем при создании своих программ. Современные реализации, чаще всего, соответствуют стандарту MPI версии 1.1. В 1997—1998 годах появился стандарт MPI-2.0, значительно расширивший функциональность предыдущей версии. Однако до сих пор этот вариант MPI не получил широкого распространения и в полном объеме не реализован ни на одной системе.

MPI поддерживает работу с языками Фортран и С. Однако это совершенно не является принципиальным, поскольку основные идеи MPI и правила оформления отдельных конструкций для этих языков во многом схожи. Полная версия интерфейса содержит описание более 125 процедур и функций. Информацию об интерфейсе MPI можно найти на тематической странице Информационно-аналитического центра по параллельным вычислениям в сети Интернет <http://parallel.ru/tech/tech-dev/mpi.html>.

Интерфейс MPI поддерживает создание параллельных программ в стиле MIMD (Multiple Instruction Multiple Data), что подразумевает объединение процессов с различными исходными текстами. Однако писать и проводить отладку таких программ очень сложно, поэтому на практике программисты чаще используют SPMD-модель (Single Program Multiple Data) параллельного программирования, в рамках которой для всех параллельных процессов используется один и тот же код. В настоящее время все больше и больше реализаций MPI поддерживают работу с нитями.

Поскольку MPI является библиотекой, то при компиляции программы необходимо присоединить соответствующие библиотечные модули. Это можно сделать в командной строке или воспользоваться предусмотренными в большинстве систем командами или скриптами mpicc (для программ на языке С), mpicc (для программ на языке С++), mpif 77/mpif 90 (для программ на языках Фортран 77/90). Опция компилятора «-o name» позволяет задать имя name для получаемого

выполнимого файла, по умолчанию выполнимый файл называется a.out, например:

```
mpif77 -o program program.f
```

После получения выполнимого файла необходимо запустить его на требуемом количестве процессоров. Для этого обычно предоставляется команда запуска MPI-приложений mpirun, например:

```
mpirun -np N <программа с аргументами>,
```

где N – число процессов, которое должно быть не более разрешенного в данной системе числа процессов для одной задачи. После запуска одна и та же программа будет выполняться всеми запущенными процессами, результат выполнения в зависимости от системы будет выдаваться на терминал или записываться в файл с предопределенным именем.

Все дополнительные объекты: имена процедур, константы, предопределенные типы данных и т.п., используемые в MPI, имеют префикс MPI\_. Если пользователь не будет использовать в программе имена с таким префиксом, то конфликтов с объектами MPI заведомо не будет. В языке С, кроме того, является существенным регистр символов в названиях функций. Обычно в названиях функций MPI первая буква после префикса MPI\_ пишется в верхнем регистре, последующие буквы – в нижнем регистре, а названия констант MPI записываются целиком в верхнем регистре. Все описания интерфейса MPI собраны в файле mpif.h (mpi.h), поэтому в начале MPI-программы должна стоять директива include 'mpif.h' (include "mpi.h" для программ на языке С).

MPI-программа – это множество параллельных взаимодействующих процессов. Все процессы порождаются один раз, образуя параллельную часть программы. В ходе выполнения MPI-программы порождение дополнительных процессов или уничтожение существующих не допускается (в MPI-2.0 такая возможность появилась). Каждый процесс работает в своем адресном пространстве, никаких общих переменных или данных в MPI нет. Основным способом взаимодействия между процессами является явная посылка сообщений.

Для локализации взаимодействия параллельных процессов программы можно создавать группы процессов, предоставляя им отдельную среду для общения – коммуникатор. Состав образуемых групп произволен. Группы могут полностью совпадать, входить одна в другую, не пересекаться или пересекаться частично. Процессы могут

взаимодействовать только внутри некоторого коммуникатора. Сообщения, отправленные в разных коммуникаторах, не пересекаются и не мешают друг другу. Коммуникаторы имеют в языке Фортран тип INTEGER (в языке С – предопределенный тип MPI\_Comm).

При старте программы всегда считается, что все порожденные процессы работают в рамках всеобъемлющего коммуникатора, имеющего предопределенное имя MPI\_COMM\_WORLD. Этот коммуникатор существует всегда и служит для взаимодействия всех запущенных процессов MPI-программы. Кроме него при старте программы имеется коммуникатор MPI\_COMM\_SELF, содержащий только один текущий процесс, а также коммуникатор MPI\_COMM\_NULL, не содержащий ни одного процесса.

Каждый процесс MPI-программы имеет в каждой группе, в которую он входит, уникальный атрибут – номер процесса, который является целым неотрицательным числом. С помощью этого атрибута происходит значительная часть взаимодействия процессов между собой. Ясно, что в одном и том же коммуникаторе все процессы имеют различные номера. Но поскольку процесс может одновременно входить в разные коммуникаторы, то его номер в одном коммуникаторе может отличаться от его номера в другом. Отсюда становятся понятными два основных атрибута процесса: коммуникатор и номер в коммуникаторе. Если группа содержит  $n$  процессов, то номер любого процесса в данной группе лежит в пределах от 0 до  $n - 1$ .

Основным способом общения процессов между собой является явная посылка сообщений. Сообщение – это набор данных некоторого типа. Каждое сообщение имеет несколько атрибутов, в частности, номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения и другие. Одним из важных атрибутов сообщения является его идентификатор или тэг. По идентификатору процесс, принимающий сообщение, например, может различить два сообщения, пришедшие к нему от одного и того же процесса. Сам идентификатор сообщения является целым неотрицательным числом, лежащим в диапазоне от 0 до MPI\_TAG\_UP, причем гарантируется, что MPI\_TAG\_UP не меньше 32767. Для работы с атрибутами сообщений введен массив (в языке С – структура), элементы которого дают доступ к их значениям.

В последнем аргументе (в языке С – в возвращаемом значении функции) большинство процедур MPI возвращают информацию об успешности завершения. В случае успешного выполнения возвращается значение MPI\_SUCCESS, иначе – код ошибки. Вид ошибки,

которая произошла при выполнении процедуры, можно определить из ее описания. Предопределенные значения, соответствующие различным ошибочным ситуациям, перечислены в файле `mpi.h`.

### 3. Парные межпроцессные обмены

#### Операции блокирующей передачи и блокирующего приема

##### Блокирующая передача

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

IN	buf	начальный адрес буфера посылки сообщения (альтернатива)
IN	count	число элементов в буфере посылки (неотрицательное целое)
IN	datatype	тип данных каждого элемента в буфере посылки (дескриптор)
IN	dest	номер процесса-получателя (целое)
IN	tag	тег сообщения (целое)
IN	comm	коммуникатор (дескриптор)

Блокирующая посылка сообщения с идентификатором `gtag`, состоящего из `count` элементов типа `datatype`, процессу с номером `dest`. Все элементы сообщения расположены подряд в буфере `buf`. Значение `count` может быть нулем. Тип передаваемых элементов `datatype` должен указываться с помощью предопределенных констант типа. Разрешается передавать сообщение самому себе.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу `dest`, остается за MPI. Следует специально отметить, что возврат из подпрограммы `MPI_Send` не означает ни того, что сообщение уже передано процессу `dest`, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший `MPI_Send`.

##### Блокирующий прием

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

OUT	buf	начальный адрес буфера процесса-получателя (альтернатива)
-----	-----	---

IN	count	число элементов в принимаемом сообщении (целое)
IN	datatype	тип данных каждого элемента сообщения (дескриптор)
IN	source	номер процесса-отправителя (целое)
IN	tag	тег сообщения (целое)
IN	comm	коммуникатор (дескриптор)
OUT	status	статус (параметры) принятого сообщения (статус)

Прием сообщения с идентификатором *tag* от процесса *source* с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения *count*. Если число принятых элементов меньше значения *count*, то гарантируется, что в буфере *buf* изменяется только элементы, соответствующие элементам принятого сообщения. Если нужно узнать точное число элементов в сообщении, то можно воспользоваться подпрограммой *MPI\_Probe*.

Блокировка гарантирует, что после возврата из подпрограммы все элементы сообщения приняты и расположены в буфере *buf*.

В качестве номера процесса-отправителя можно указать предопределенную константу *MPI\_ANY\_SOURCE* – признак того, что подходит сообщение от любого процесса. В качестве идентификатора принимаемого сообщения можно указать константу *MPI\_ANY\_TAG* – признак того, что подходит сообщение с любым идентификатором.

Если процесс посыпает два сообщения другому процессу и оба эти сообщения соответствуют одному и тому же вызову *MPI\_Recv*, то первым будет принято то сообщение, которое было отправлено раньше.

#### Возвращаемая статусная информация

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
int *count)
```

IN	status	возвращает статус операции приема (статус)
IN	datatype	тип данных каждого элемента приемного буфера (дескриптор)
OUT	count	количество полученных элементов (целое)

По значению параметра *status* данная подпрограмма определяет число уже принятых (после обращения к *MPI\_Recv*) или принимаемых (после обращения к *MPI\_Probe* или *MPI\_Iprobe*) элементов сообщения типа *datatype*. Аргумент *datatype* передается в *MPI\_GET\_COUNT*, чтобы

улучшить характеристики обмена. Сообщение может быть получено без подсчета числа элементов, которое оно содержит, и этот подсчет часто не нужен. Становится возможным использовать ту же самую функцию после вызова MPI\_PROBE.

## Коммуникационные режимы

### Буферизованный режим посылки с блокировкой

```
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)
```

IN	buf	начальный адрес буфера посылки (альтернатива)
IN	count	число элементов в буфере посылки (неотрицательное целое)
IN	datatype	тип данных каждого элемента в буфере посылки (дескриптор)
IN	dest	номер процесса-получателя (целое)
IN	tag	тег сообщения (целое)
IN	comm	коммуникатор (дескриптор)

Буферизованный (buffered) режим операции посылки может стартовать вне зависимости от того, инициирован ли соответствующий прием. Однако, в отличие от стандартной посылки, эта операция является локальной и ее завершение не зависит от обстоятельств приема. Следовательно, если посылка выполнена и никакого соответствующего приема не инициировано, тогда MPI обязан буферизовать исходящее сообщение, чтобы позволить завершиться вызову send. Если не имеется достаточного объема буферного пространства, возникнет ошибка. Объем буферного пространства задается пользователем. Для того, чтобы буферизованный режим был эффективным, может потребоваться распределение буферов пользователем.

### Синхронный режим посылки с блокировкой

```
int MPI_Ssend(void* buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)
```

IN	buf	начальный адрес буфера посылки (альтернатива)
IN	count	число элементов в буфере посылки (неотрицательное целое)
IN	datatype	тип данных каждого элемента в буфере посылки (дескриптор)

IN	<code>dest</code>	номер процессса-получателя (целое)
IN	<code>tag</code>	тег сообщения (целое)
IN	<code>comm</code>	коммуникатор (дескриптор)

Посылка, которая использует синхронный (synchronous) режим, может стартовать вне зависимости от того, был ли начат соответствующий прием. Однако посылка будет завершена успешно только тогда, когда соответствующая операция приема стартовала. Следовательно, завершение синхронной передачи не только указывает, что буфер отправителя может быть повторно использован, но также и отмечает то, что получатель достиг определенной точки в своей работе, а именно, что он начал выполнение приема. Если и посылка, и прием являются блокирующими операциями, тогда использование синхронного режима обеспечивает синхронную коммуникационную семантику: посылка не завершается на любой стороне обмена, пока оба процесса не выполнят randevu в процессе операции обмена. Выполнение обмена в этом режиме является нелокальным.

#### Режим посылки по готовности с блокировкой

```
int MPI_Rsend(void* buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)
```

IN	<code>buf</code>	начальный адрес буфера посылки (альтернатива)
IN	<code>count</code>	число элементов в буфере посылки (неотрицательное целое)
IN	<code>datatype</code>	тип данных каждого элемента в буфере посылки (дескриптор)
IN	<code>dest</code>	номер процессса-получателя (целое)
IN	<code>tag</code>	тег сообщения (целое)
IN	<code>comm</code>	коммуникатор (дескриптор)

Посылка, которая использует режим обмена по готовности (ready communication mode), может быть запущена только тогда, когда прием уже инициирован. В противном случае операция считается ошибочной и результат будет неопределенным. На некоторых системах обмен по готовности позволяет устраниить необходимость в randevu, что улучшает характеристики обмена. Завершение операции посылки не зависит от состояния приема и в основном указывает на то, что буфер посылки может быть повторно использован. Операция посылки, которая использует режим готовности, имеет ту же семантику, как и стандартная или синхронная передача; это означает, что отправитель обеспечивает систему дополнительной информацией (а именно, что прием уже

инициирован), которая может уменьшить накладные расходы. Вследствие этого в правильной программе посылка по готовности может быть замещена стандартной передачей без влияния на поведение программы (но не на характеристики).

#### Инициация неблокирующего обмена

int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)	начальный адрес буфера посылки
IN       buf	(альтернатива)
IN       count	число элементов в буфере посылки (целое)
IN       datatype	тип каждого элемента в буфере посылки (дескриптор)
IN       dest	номер процесса-получателя (целое)
IN       tag	тэг сообщения (целое)
IN       comm	коммуникатор (дескриптор)
OUT      request	запрос обмена (дескриптор)

Рассматриваемая передача сообщения аналогична *MPI\_Send*. Однако возврат из подпрограммы происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере *buf*. Это означает, что нельзя повторно использовать данный буфер для других целей без получения дополнительной информации о завершении данной посылки. Окончание процесса передачи (т.е. того момента, когда можно переиспользовать буфер *buf* без опасения испортить передаваемое сообщение) можно определить с помощью параметра *request* и процедур *MPI\_Wait* и *MPI\_Test*. Сообщение, отправленное любой из процедур *MPI\_Send* и *MPI\_Isend*, может быть принято любой из процедур *MPI\_Recv* и *MPI\_Irecv*. Неблокирующий вызов посылки указывает на то, что система может стартовать, копируя данные из буфера отправителя. Отправитель не должен обращаться к любой части буфера посылки после того, как вызвана операция неблокируемой передачи, пока посылка не завершится.

#### Распределение и использование буферов

int MPI_Buffer_attach( void* buffer, int size)	начальный адрес буфера
IN       buffer	(альтернатива)
IN       size	размер буфера в байтах (целое)

Предусмотренный в MPI буфер в памяти пользователя применяется для буферизации исходящих сообщений. Буфер используется только

сообщениями, посланными в буферизованном режиме. За одну операцию к процессу может быть присоединен только один буфер.

```
int MPI_Buffer_detach( void* buffer, int* size)
OUT      buffer_addr          Начальный адрес буфера
          (альтернатива)
OUT      size                 Размер буфера, в байтах (целое)
```

Отключение буфера операционно связано с MPI. Вызов возвращает адрес и размер отключенного буфера. Эта операция будет блокирована до тех пор, пока находящееся в буфере сообщение не будет передано. После выполнения этой функции пользователь может повторно использовать или перераспределять объем памяти, занятый буфером.

### Неблокирующий обмен

#### Инициация буферизованного обмена без блокировки

```
int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

#### Инициация синхронного обмена без блокировки

```
int MPI_Issend(void* buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

#### Инициация обмена без блокировки по готовности

```
int MPI_Irsend(void* buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

#### Инициация неблокирующего приема

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Request *request)
```

OUT	buf	начальный адрес буфера процесса-получателя (альтернатива)
IN	count	число элементов в принимаемом сообщении (целое)
IN	datatype	тип данных каждого элемента сообщения (дескриптор)
IN	source	номер процесса-отправителя (целое)
IN	tag	тэг сообщения (целое)
IN	comm	коммуникатор (дескриптор)
OUT	status	статус (параметры) принятого сообщения (статус)

Прием сообщения аналогичен *MPI\_Recv*, однако возврат из подпрограммы происходит сразу после инициализации процесса приема без ожидания получения сообщения в буфере *buf*. Окончание процесса приема можно определить с помощью параметра *request* и процедур *MPI\_Wait* и *MPI\_Test*.

#### Завершение неблокирующего обмена

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

INOUT	request	запрос (дескриптор)
OUT	status	объект состояния (статус)

В данном случае происходит ожидание завершения асинхронных процедур *MPI\_Isend* или *MPI\_Irecv*, ассоциированных с идентификатором *request*. В случае приема атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра *status*. Обращение к *MPI\_WAIT* заканчивается, когда завершена операция, указанная в запросе. Если коммуникационный объект, связанный с этим запросом, был создан вызовом неблокирующей посылки или приема, тогда этот объект удаляется при обращении к *MPI\_WAIT* и дескриптор запроса устанавливается в *MPI\_REQUEST\_NULL*. *MPI\_WAIT* является нелокальной операцией. Разрешается вызывать *MPI\_WAIT* с нулевым или неактивным аргументом запроса. В этом случае операция заканчивается немедленно со статусом *empty*.

#### Проверка завершения посылки или приема

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

INOUT	request	коммуникационный (дескриптор)	запрос
OUT	flag	true, если операция завершена (логический тип)	
OUT	status	статусный объект (статус)	

При таком вызове осуществляется проверка завершенности асинхронных процедур *MPI\_Isend* или *MPI\_Irecv*, ассоциированных с идентификатором *request*. Если завершена процедура приема, то атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра *status*. Обращение к *MPI\_TEST* возвращает *flag = true*, если операция, указанная в запросе, завершена. В этом случае

статусный объект содержит информацию о завершенной операции; если коммуникационный объект был создан неблокирующей посылкой или приемом, то он входит в состояние дедлока и обработка запроса устанавливается в MPI\_REQUEST\_NULL. Другими словами, вызов возвращает flag = false. В этом случае значение статуса не определено. MPI\_TEST является локальной операцией. Можно вызывать MPI\_TEST с нулевым или неактивным аргументом запроса. В таком случае операция возвращает flag = true и empty для status.

#### Освобождение объекта коммуникационного запроса

```
int MPI_Request_free(MPI_Request *request)
```

#### Множественные завершения

Ожидание завершения любой из описанных функций посылки или приема

```
int MPI_Waitany(int count, MPI_Request *array_of_requests,
int *index, MPI_Status *status)
IN      count           длина списка (целое)
INOUT   array_of_requests  массив запросов (массив
                           дескрипторов)
OUT     index           индекс дескриптора для
                           завершенной операции (целое)
OUT     status           статусный объект (статус)
```

Выполнение процесса блокируется до тех пор, пока какая-либо операция обмена, ассоциированная с указанными идентификаторами, не будет завершена. Если несколько операций могут быть завершены, то случайным образом выбирается одна из них. Параметр *index* содержит номер элемента в массиве *requests*, содержащего идентификатор завершенной операции. Если запрос был создан операцией неблокирующего обмена, тогда он удаляется и дескриптор запроса устанавливается в MPI\_REQUEST\_NULL. Список *array\_of\_request* может содержать нуль или неактивные дескрипторы. Если список не содержит активных дескрипторов (список имеет нулевую длину или все элементы являются нулями или неактивны), тогда вызов заканчивается немедленно с *index* = MPI\_UNDEFINED и со статусом empty.

#### Проверка завершения любой ранее начатой операции

```
int MPI_Testany(int count, MPI_Request *array_of_requests,
int *index, int *flag, MPI_Status *status)
IN      count           длина списка (целое)
INOUT   array_of_requests  массив запросов (массив
                           дескрипторов)
```

OUT	<i>index</i>	индекс дескриптора для завершенной операции (целое)
OUT	<i>flag</i>	true, если одна из операций завершена (логический тип)
OUT	<i>status</i>	статусный объект (статус)

Если к моменту вызова подпрограммы хотя бы одна из операций обмена завершилась, то в параметре *flag* возвращается значение *1*, *index* содержит номер соответствующего элемента в массиве *requests*, а *status* – параметры сообщения. Массив может содержать нуль или неактивные дескрипторы. Если массив не содержит активных дескрипторов, тогда вызов заканчивается немедленно с *flag* = *true*, *index* = *MPI\_UNDEFINED*, и *status* = *empty*.

#### Ожидание завершения всех обменов

int	<i>MPI_Waitall</i> (int <i>count</i> , MPI_Request *array_of_requests, MPI_Status *array_of_statuses)	
IN	<i>count</i>	длина списков (целое)
INOUT	<i>array_of_requests</i>	массив запросов (массив дескрипторов)
OUT	<i>array_of_statuses</i>	массив статусных объектов (массив статусов)

Функция блокирует работу, пока все операции обмена, связанные с активными дескрипторами в списке, не завершатся, и возвращает статус всех операций (это включает случай, когда в списке нет активных дескрипторов). Когда один или более обменов, завершенных обращением к *MPI\_WAITALL*, оказались неудачны, желательно возвратить специальную информацию по каждому обмену. Функция *MPI\_WAITALL* возвращает в таком случае код *MPI\_ERR\_IN\_STATUS* и устанавливает в поля ошибки каждого статуса специфический код ошибки. Этот код равен *MPI\_SUCCESS*, если обмен завершен, или другому значению, если обмен не состоялся; он может иметь значение *MPI\_ERR\_PENDING*, если он и не завершен, и не в состоянии отказа. Функция *MPI\_WAITALL* будет возвращать *MPI\_SUCCESS*, если никакой из запросов не вызвал ошибку или будет возвращать специальный код ошибки, если запрос не выполнился по другим причинам (таким, как неверный аргумент). В таком случае функция не будет корректировать поле ошибки в статусе.

#### Проверка завершения всех ранее начатых операций обмена

int	<i>MPI_Testall</i> (int <i>count</i> , MPI_Request *array_of_requests, int *flag, MPI_Status *array_of_statuses)	
-----	---	--

IN	count	длина списка (целое)
INOUT	array_of_requests	массив запросов (массив дескрипторов)
OUT	flag	(логический тип)
OUT	array_of_statuses	массив статусных объектов (массив статусов)

Функция возвращает flag = true, если обмены, связанные с активными дескрипторами в массиве, завершены (это включает случай, когда в списке нет активных дескрипторов). В этом случае каждый статусный элемент, который соответствует активному дескриптору, устанавливается в статус соответствующего обмена; если запрос был создан вызовом неблокирующего обмена, то он удаляется и дескриптор устанавливается в MPI\_REQUEST\_NULL. Каждый статусный элемент, который соответствует нулю или неактивному дескриптору, устанавливается в состояние empty. В противном случае возвращается flag = false, никакие запросы не модифицируются и значения статусных элементов неопределены. Операция является локальной. Ошибки, которые имеют место при выполнении MPI\_TESTALL, обрабатываются также, как и в случае с функцией MPI\_WAITALL.

#### Ожидание завершения некоторых заданных обменов

IN	incount	длина массива запросов (целое)
INOUT	array_of_requests	массив запросов (массив дескрипторов)
OUT	outcount	число завершенных запросов (целое)
OUT	array_of_indices	массив индексов операций, которые завершены (массив целых)
OUT	array_of_statuses	массив статусных операций для завершенных операций (массив статусов)

Выполнение процесса блокируется до тех пор, пока по крайней мере одна из операций обмена, ассоциированных с указанными идентификаторами, не будет завершена. Параметр *outcount* содержит число завершенных операций, а первые *outcount* элементов массива *indexe*s содержат номера элементов массива *requests* с их идентификаторами. Первые *outcount* элементов массива *statuses* содержат параметры завершенных операций. Если один или более обменов,

завершенных MPI\_WAITsome, не могут быть выполнены, то надо возвращать по каждому обмену специфическую информацию. При этом аргументы `outcount`, `array_of_indices` и `array_of_statuses` будут индицировать завершение всех обменов, успешных или неуспешных. Вызов будет возвращать код ошибки MPI\_ERR\_IN\_STATUS и устанавливать поле ошибки каждого возвращенного статуса, чтобы указать на успешное завершение или возвратить специфический код ошибки. Вызов будет возвращать MPI\_SUCCESS, если ни один запрос не содержал ошибки, и специальный код ошибки, если запрос не может быть выполнен по какой-то причине (такой, как неправильный аргумент). В таких случаях поля ошибок статуса не будут корректироваться.

#### Проверка завершения заданных операций

	<code>int MPI_Testsome(int incount, MPI_Request *array_of_requests,</code>			
	<code>int *outcount, int *array_of_indices, MPI_Status</code>			
	<code>*array_of_statuses)</code>			
IN	<code>incount</code>	длина массива	запросов	
		(целое)		
INOUT	<code>array_of_requests</code>	массив	запросов (массив	
		дескрипторов)		
OUT	<code>outcount</code>	число завершенных	запросов	
		(целое)		
OUT	<code>array_of_indices</code>	массив	индексов	
		завершенных	операций (массив	
		целых)		
OUT	<code>array_of_statuses</code>	массив	объектов	
		завершенных	операций (массив	
		статусов)		

Функция MPI\_TESTsome ведет себя подобно MPI\_WAITsome за исключением того, что заканчивается немедленно. Если ни одной операции не завершено, она возвращает `outcount = 0`. Если не имеется активных дескрипторов в списке, она возвращает `outcount = MPI_UNDEFINED`. MPI\_TESTsome является локальной операцией, которая заканчивается немедленно, тогда как MPI\_WAITsome будет блокировать процесс до завершения обменов, если в списке содержится хотя бы один активный дескриптор. Оба вызова выполняют требование однозначности: если запрос на прием повторно появляется в списке запросов, передаваемых MPI\_WAITsome или MPI\_TESTsome, и соответствующая посылка была инициирована, тогда прием будет рано или поздно завершен успешно, если передача не закрыта другим приемом;

аналогичная ситуация для запросов посылок. Ошибки при выполнении MPI\_TESTsome обрабатываются так же, как и для MPI\_WAITsome.

### Проба и отмена

#### Неблокирующий тест сообщения

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,  
MPI_Status *status)
```

#### Блокирующий тест сообщения

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status  
*status)
```

#### Отмена коммуникационного запроса

```
int MPI_Cancel(MPI_Request *request)
```

#### Проверка отмены запроса

```
int MPI_Test_cancelled(MPI_Status *status, int *flag)
```

### Персистентные коммуникационные запросы

Часто запрос с одним и тем же набором аргументов повторно выполняется во внутреннем цикле параллельных вычислений. В такой ситуации можно оптимизировать обмен путем однократного включения списка аргументов в персистентный (persistent) коммуникационный запрос и повторного использования этого запроса для инициации и завершения обмена.

Персистентный запрос, созданный таким образом, может восприниматься как коммуникационный порт или «полуканал». Он не обеспечивает полноты функций обычного канала, поскольку нет никакой связи между передающим и приемным портами. Эта конструкция позволяет уменьшить накладные расходы на коммуникацию между процессом и контроллером, но не расходы на коммуникацию между одним контроллером и другим.

Если сообщение послано на основе персистентного запроса, то оно не обязательно должно быть принято операцией, также использующей персистентный запрос, и наоборот.

**Создание дескриптора для стандартной посылки**

```
int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

**Создание дескриптора для буферизованной посылки**

```
int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

**Создание дескриптора для синхронной передачи**

```
int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

**Создание дескриптора для посылки по готовности**

```
int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

**Создание дескриптора для приема**

```
int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
```

**Инициация обмена с персистентным дескриптором запросов**

```
int MPI_Start(MPI_Request *request)
```

**Запуск совокупности запросов**

```
int MPI_Startall(int count, MPI_Request *array_of_requests)
```

**Совмещенные прием и передача сообщений (send-receive)**

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, MPI_Datatype recvtag, MPI_Comm comm, MPI_Status *status)
```

IN	sendbuf	начальный адрес буфера отправителя (альтернатива)
IN	sendcount	число элементов в буфере отправителя (целое)
IN	sendtype	тип элементов в буфере отправителя (дескриптор)
IN	dest	номер процесса-получателя (целое)
IN	sendtag	тег процесса-отправителя (целое)
OUT	recvbuf	начальный адрес приемного буфера (альтернатива)

IN	<code>recvcount</code>	число элементов в приемном буфере (целое)
IN	<code>recvtype</code>	тип элементов в приемном буфере (дескриптор)
IN	<code>source</code>	номер процесса-отправителя (целое)
IN	<code>recvtag</code>	тег процесса-получателя (целое)
IN	<code>comm</code>	коммуникатор (дескриптор)
OUT	<code>status</code>	статус (статус)

Функция MPI\_SENDRECV выполняет операции блокирующей передачи и приема. Передача и прием используют тот же самый коммуникатор, но возможно различные тэги. Буфера отправителя и получателя должны быть разделены и могут иметь различную длину и типы данных.

#### Посылка и прием сообщений с использованием одного буфера

```
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype
datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm
comm, MPI_Status *status)
```

#### Производные типы данных

##### Создание непрерывного типа данных

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
MPI_Datatype *newtype)
```

##### Создание векторного типа данных

```
int MPI_Type_vector(int count, int blocklength, int stride,
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

##### Создание индексированного типа данных

```
int MPI_Type_indexed(int count, int *array_of_blocklengths,
int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype
*newtype)
```

##### Создание нового типа данных

```
int MPI_Type_struct(int count, int *array_of_blocklengths,
MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
MPI_Datatype *newtype)
```

IN	<code>count</code>	число блоков (целое)
IN	<code>array_of_blocklength</code>	число элементов в каждом блоке (массив целых)
IN	<code>array_of_displacements</code>	смещение каждого блока в байтах (массив целых)
IN	<code>array_of_types</code>	тип элементов в каждом

OUT	<code>newtype</code>	блоке (массив дескрипторов объектов типов данных) новый тип данных (дескриптор)
-----	----------------------	--

`MPI_TYPE_STRUCT` является наиболее общим типом конструктора. Он отличается от предыдущего тем, что позволяет каждому блоку состоять из репликаций различного типа. Предпочтение отдается функции `MPI_TYPE_CREATESTRUCT`.

#### Получение адреса в ячейке памяти

```
int MPI_Address(void* location, MPI_Aint *address)
```

#### Создание экстента типа данных

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```

#### Определение числа байтов с занятymi элементами типа данных

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

IN	<code>datatype</code>	тип (дескриптор)	данных
OUT	<code>size</code>	размер типа (целое)	данных

Функция `MPI_TYPE_SIZE` возвращает общий размер в байтах элементов в сигнатуре типа, связанный с `datatype`; это общий размер данных в сообщении, которое было бы создано с этим типом данных. Элементы, которые появляются в типе данных несколько раз, подсчитываются с учетом их кратности.

#### Определение нижней границы типа данных

```
int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint* displacement)
```

#### Определение верхней границы типа данных

```
int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)
```

#### Объявление типа данных

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

#### **Отмечание объекта типа данных для удаления**

```
int MPI_Type_free(MPI_Datatype *datatype)
```

### Определение номера базового элемента в типе данных

```
int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype, int *count)
```

## Упаковка данных в непрерывный буфер

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype,
void *outbuf, int outsize, int *position, MPI_Comm comm)
```

## Распаковка данных из непрерывного буфера

```
int MPI_Unpack(void* inbuf, int insize, int *position, void  
*outbuf, int outcount, MPI Datatype datatype, MPI Comm comm)
```

## Определение необходимого размера для упаковки типа данных

```
    int MPI_Pack_size(int incount, MPI_Datatype datatype,  
MPI_Comm comm, int *size)
```

Ошибочные ситуации перечислены в файле `mpi.h`.

#### *4. Операции коллективного обмена*

## Барьерная синхронизация

```
int MPI Barrier(MPI_Comm comm )
```

**IN** коммуникатор (дескриптор)

Функция барьерной синхронизации MPI\_BARRIER блокирует вызывающий процесс, пока все процессы группы не вызовут её. В каждом процессе управление возвращается только тогда, когда все процессы в группе вызовут процедуру.

## Широковещательный обмен

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
int root, MPI_Comm comm )
```

INOUT	buffer	адрес (альтернатива)	начала	буфера
IN	count	количество (целое)	записей в	буфере
IN	datatype	тип (дескриптор)	данных в	буфере
IN	root	номер	корневого	процесса

IN **сомн** (целое) коммуникатор (дескриптор)

Функция широковещательной передачи MPI\_BCAST посылает сообщение из корневого процесса всем процессам группы, включая себя. Она вызывается всеми процессами группы с одинаковыми аргументами для comm и root. В момент возврата управления содержимое корневого буфера обмена будет уже скопировано во все процессы. В аргументе datatype можно задавать производные типы данных. Сигнатура типа данных count, datatype любого процесса обязана совпадать с соответствующей сигнатурой в корневом процессе. Необходимо, чтобы количество посланных и полученных данных совпадало попарно для корневого и каждого другого процессов. Такое ограничение имеют и все остальные коллективные операции, выполняющие перемещение данных. Однако по-прежнему разрешается различие в картах типов данных между отправителями и получателями.

## Сборка данных

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

IN	sendbuf	начальный адрес буфера процессора отправителя (альтернатива)
IN	sendcount	количество элементов в отсылаемом сообщении (целое)
IN	sendtype	тип элементов в отсылаемом сообщении (дескриптор)
OUT	recvbuf	начальный адрес буфера процесса сборки данных (альтернатива, существенно только для корневого процесса)
IN	recvcount	количество элементов в принимаемом сообщении (целое, имеет значение только для корневого процесса)
IN	recvtype	тип данных элементов в буфере процесса-получателя (дескриптор)
IN	root	номер процесса-получателя (целое)
IN	comm	коммуникатор (дескриптор)

При выполнении операции сборки данных MPI\_GATHER каждый процесс, включая корневой, посыпает содержимое своего буфера в корневой процесс. Корневой процесс получает сообщения, располагая их в порядке возрастания номеров процессов. В общем случае как для sendtype, так и для recvtype разрешены производные типы данных. Сигнатура типа данных sendcount, sendtype у процесса i должна быть

такой же, как сигнатура recvcount, recvtype корневого процесса. Это требуется для того, чтобы количество посланных и полученных данных совпадало попарно для корневого и каждого другого процессов. Однако по-прежнему разрешается различие в картах типов между отправителями и получателями. В корневом процессе используются все аргументы функции, в то время как у остальных процессов используются только аргументы sendbuf, sendcount, sendtype, root, comm. Аргументы comm и root должны иметь одинаковые значения во всех процессах. Описанные в функции MPI\_GATHER количества и типы данных не должны являться причиной того, чтобы любая ячейка корневого процесса записывалась бы более одного раза. Такой вызов является неверным.

#### Сборка данных с обеспечением переменного числа данных

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int *recvcounts, int *displs,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

IN	sendbuf	начальный адрес буфера процесса-отправителя (альтернатива)
IN	sendcount	количество элементов в отсылаемом сообщении (целое)
IN	sendtype	тип элементов в отсылаемом сообщении (дескриптор)
OUT	recvbuf	начальный адрес буфера процесса сборки данных (альтернатива, существенно только для корневого процесса)
IN	recvcounts	массив целых чисел (по размеру группы), содержащий количества элементов, которые получены от каждого из процессов (используется только корневым процессом)
IN	displs	массив целых чисел (по размеру группы). Элемент i определяет смещение относительно recvbuf, в котором размещаются данные из процесса i (используется только корневым процессом)
IN	recvtype	тип данных элементов в буфере процесса-получателя (дескриптор)
IN	root	номер процесса-получателя (целое)
IN	comm	коммуникатор (дескриптор)

По сравнению с MPI\_GATHER при использовании функции MPI\_GATHERV разрешается принимать от каждого процесса переменное число элементов данных, поэтому в функции MPI\_GATHERV аргумент recvcount является массивом. Она также обеспечивает большую гибкость в

размещении данных в корневом процессе. Для этой цели используется новый аргумент `displs`. В корневом процессе используются все аргументы функции `MPI_GATHERV`, а на всех других процессах используются только аргументы `sendbuf`, `sendcount`, `sendtype`, `root`, `comm`. Переменные `comm` и `root` должны иметь одинаковые значения во всех процессах. Описанные в функции `MPI_GATHERV` количества, типы данных и смещения не должны приводить к тому, чтобы любая область корневого процесса записывалась бы более одного раза. Такой вызов является неверным.

### Рассылка

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int
root, MPI_Comm comm)
```

IN	<code>sendbuf</code>	начальный адрес буфера рассылки (альтернатива, используется только корневым процессом)
IN	<code>sendcount</code>	количество элементов, посылаемых каждому процессу (целое, используется только корневым процессом)
IN	<code>sendtype</code>	тип данных элементов в буфере посылки (дескриптор, используется только корневым процессом)
OUT	<code>recvbuf</code>	адрес буфера процесса-получателя (альтернатива)
IN	<code>recvcount</code>	количество элементов в буфере корневого (целое)
IN	<code>recvtype</code>	тип данных элементов приемного буфера (дескриптор)
IN	<code>root</code>	номер процесса-получателя (целое)
IN	<code>comm</code>	коммуникатор (дескриптор)

Буфер отправки игнорируется всеми некорневыми процессами. Сигнатура типа, связанная с `sendcount`, `sendtype`, должна быть одинаковой для корневого процесса и всех других процессов (хотя карты типов могут быть разными). Необходимо, чтобы количество посланных и полученных данных совпадало попарно для корневого и каждого другого процессов. Однако по-прежнему разрешается различие в картах типов между отправителями и получателями. Корневой процесс использует все аргументы функции, а другие процессы используют только аргументы `recvbuf`, `recvcount`, `recvtype`, `root`, `comm`. Аргументы `root` и `comm` должны быть одинаковыми во всех процессах. Описанные в функции `MPI_SCATTER` количества и типы данных не должны являться причиной

того, чтобы любая ячейка корневого процесса записывалась бы более одного раза. Такой вызов является неверным.

Рассылка части буфера одного процесса всем процессам в группе

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,  
MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype  
recvtype, int root, MPI_Comm comm)
```

IN	sendbuf	адрес буфера (альтернатива, используется корневым процессом)	посылки только
IN	sendcounts	целочисленный массив (размера группы), определяющий число элементов, для отправки каждому процессу	
IN	displs	целочисленный массив (размера группы). Элемент i указывает смещение (относительно sendbuf, из которого берутся данные для процесса take the i)	
IN	sendtype	тип элементов посылающего буфера (дескриптор)	
OUT	recvbuf	адрес принимающего буфера (альтернатива)	
IN	recvcount	число элементов в посылающем буфере (целое)	
IN	recvtype	тип данных элементов принимающего буфера (дескриптор)	
IN	root	номер посылающего процесса (целое)	
IN	comm	коммуникатор (дескриптор)	

По сравнению с MPI\_GATHER при использовании функции MPI\_GATHERV разрешается принимать от каждого процесса переменное число элементов данных, поэтому в функции MPI\_GATHERV аргумент recvcount является массивом. Она также обеспечивает большую гибкость в размещении данных в корневом процессе. Для этой цели используется новый аргумент displs. Сообщения помещаются в принимающий буфер корневого процесса в порядке возрастания их номеров, то есть данные, посланные процессом j, помещаются в j-ю часть принимающего буфера recvbuf на корневом процессе. j-я часть recvbuf начинается со смещения displs[j]. Номер принимающего буфера игнорируется во всех некорневых процессах. В корневом процессе используются все аргументы функции MPI\_GATHERV, а на всех других процессах используются только аргументы sendbuf, sendcount, sendtype, root, comm. Переменные comm и root должны иметь одинаковые значения во всех процессах. Описанные в функции MPI\_GATHERV количества, типы данных и смещения не

должны приводить к тому, чтобы любая область корневого процесса записывалась бы более одного раза. Такой вызов является неверным.

#### Сборка для всех процессов

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
MPI_Comm comm)
```

IN	sendbuf	начальный адрес посылающего буфера (альтернатива)
IN	sendcount	количество элементов в буфере (целое)
IN	sendtype	тип данных элементов в посылающем буфере (дескриптор)
OUT	recvbuf	адрес принимающего буфера (альтернатива)
IN	recvcount	количество элементов, полученных от любого процесса (целое)
IN	recvtype	тип данных элементов принимающего буфера (дескриптор)
IN	comm	коммуникатор (дескриптор)

Функцию MPI\_ALLGATHER можно представить как MPI\_GATHER, где результат принимают все процессы, а не только главный. Блок данных, посланный j-м процессом принимается каждым процессом и помещается в j-й блок буфера recvbuf. Сигнатура типа, связанная с sendcount, sendtype, должна быть одинаковой во всех процессах.

#### Сборка данных от всех процессов и поставка их всем процессам

```
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int *recvcounts, int *displs,
MPI_Datatype recvtype, MPI_Comm comm)
```

IN	sendbuf	начальный адрес посылающего буфера (альтернатива)
IN	sendcount	количество элементов в посылающем буфере (целое)
IN	sendtype	тип данных элементов в посылающем буфере (дескриптор)
OUT	recvbuf	адрес принимающего буфера (альтернатива)
IN	recvcounts	целочисленный массив (размера группы), содержащий количество элементов, полученных от каждого процесса
IN	displs	целочисленный массив (размера группы). Элемент i представляет смещение области (относительно recvbuf), где помещаются принимаемые

		данные от процесса i
IN	recvtype	тип данных элементов принимающего буфера (дескриптор)
IN	comm	коммуникатор (дескриптор)

Функцию MPI\_ALLGATHERV можно представить как MPI\_GATHERV, но при ее использовании результат получают все процессы, а не только один корневой. j-й блок данных, посланный каждым процессом, принимается каждым процессом и помещается в j-й блок буфера recvbuf. Эти блоки не обязаны быть одинакового размера. Сигнатура типа, связанного с sendcount, sendtype в процессе j, должна быть такой же, как сигнатуре типа, связанного с recvcounts[j], recvtype в любом другом процессе.

Посылка данных от всех процессов всем процессам

```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
MPI_Comm comm)
```

IN	sendbuf	начальный адрес посылающего буфера (альтернатива)
IN	sendcount	количество элементов, посылаемых в каждый процесс (целое)
IN	sendtype	тип данных элементов посылающего буфера (дескриптор)
OUT	recvbuf	адрес принимающего буфера (альтернатива)
IN	recvcount	количество элементов, принятых от какого-либо процесса (целое)
IN	recvtype	тип данных элементов принимающего буфера (дескриптор)
IN	comm	коммуникатор (дескриптор)

MPI\_ALLTOALL – это расширение функции MPI\_ALLGATHER для случая, когда каждый процесс посылает различные данные каждому получателю. j-й блок, посланный процессом i, принимается процессом j и помещается в i-й блок буфера recvbuf. Сигнатуре типа, связанная с sendcount, sendtype в каждом процессе, должна быть такой же, как и в любом другом процессе. Необходимо, чтобы количество посланных данных было равно количеству полученных данных между каждой парой процессов. Как обычно, карты типа могут отличаться. Все аргументы используются всеми процессами. Аргумент comm должен иметь одинаковое значение во всех процессах.

### Посылка данных от всех процессов всем процессам со смещением

```
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int
*sdispls, MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
```

IN	sendbuf	начальный адрес посылающего буфера (альтернатива)
IN	sendcounts	целочисленный массив (размера группы), определяющий количество посыпаемых каждому процессу элементов
IN	sdispls	целочисленный массив (размера группы). Элемент $j$ содержит смещение области (относительно <code>sendbuf</code> ), из которой берутся данные для процесса $j$
IN	sendtype	тип данных элементов посылающего буфера (дескриптор)
OUT	recvbuf	адрес принимающего буфера (альтернатива)
IN	recvcounts	целочисленный массив (размера группы), содержащий число элементов, которые могут быть приняты от каждого процесса
IN	rdispls	целочисленный массив (размера группы). Элемент $i$ определяет смещение области (относительно <code>recvbuf</code> ), в которой размещаются данные, получаемые из процесса $i$
IN	recvtype	тип данных элементов принимающего буфера (дескриптор)
IN	comm	коммуникатор (дескриптор)

`MPI_ALLTOALLV` обладает большей гибкостью, чем функция `MPI_ALLTOALL`, поскольку размещение данных на передающей стороне определяется аргументом `sdispls`, а на стороне приема – независимым аргументом `rdispls`.  $j$ -й блок, посланный процессом  $i$ , принимается процессом  $j$  и помещается в  $i$ -й блок `recvbuf`. Эти блоки не обязаны быть одного размера. Сигнатура типа, связанная с `sendcount[j]`, `sendtype` в процессе  $i$ , должна быть такой же и для процесса  $j$ . Необходимо, чтобы количество посланных данных было равно количеству полученных данных для каждой пары процессов. Карты типа для отправителя и приемника могут отличаться. Все аргументы используются всеми процессами. Значение аргумента `comm` должно быть одинаковым во всех процессах.

Выполнение глобальной операции над значениями всех процессов и возвращение результата в один процесс

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

IN	sendbuf	адрес посылающего буфера (альтернатива)
OUT	recvbuf	адрес принимающего буфера (альтернатива, используется только корневым процессом)
IN	count	количество элементов в посылающем буфере (целое)
IN	datatype	тип данных элементов посылающего буфера (дескриптор)
IN	op	операция редукции (дескриптор)
IN	root	номер главного процесса (целое)
IN	comm	коммуникатор (дескриптор)

Функция MPI\_REDUCE объединяет элементы входного буфера каждого процесса в группе, используя операцию op, и возвращает объединенное значение в выходной буфер процесса с номером root. Буфер ввода определен аргументами sendbuf, count и datatype; буфер вывода определен параметрами recvbuf, count и datatype; оба буфера имеют одинаковое число элементов одинакового типа. Функция вызывается всеми членами группы с одинаковыми аргументами count, datatype, op, root и comm. Таким образом, все процессы имеют входные и выходные буфера одинаковой длины и с элементами одного типа. Каждый процесс может содержать либо один элемент, либо последовательность элементов, в последнем случае операция выполняется над всеми элементами в этой последовательности. Например, если выполняется операция MPI\_MAX , и посылающий буфер содержит два элемента – числа с плавающей точкой (count = 2, datatype = MPI\_FLOAT), то recvbuf(1) = sendbuf(1) и recvbuf(2) = sendbuf(2).

#### Создание определенного пользователем дескриптора функции

```
int MPI_Op_create(MPI_User_function *function, int commute,
MPI_Op *op)
```

IN	function	определенная пользователем операция (функция)
IN	commute	true, если операция коммутативна; иначе false.
OUT	op	операция (дескриптор)

Функция MPI\_OP\_CREATE связывает определенную пользователем глобальную операцию с указателем op, который впоследствии может быть использован в MPI\_REDUCE, MPI\_ALLREDUCE, MPI\_SCATTER и MPI\_SCAN. Определенная пользователем операция предполагается ассоциативной. Если commute = true, то операция должна быть как коммутативной, так и ассоциативной. Если commute = false, то порядок операндов фиксирован, операнды располагаются по возрастанию номеров процессов, начиная с нулевого. Порядок оценки может быть изменен, чтобы использовать преимущество ассоциативности операции. Если commute = true, то порядок оценки может быть изменен, чтобы использовать достоинства коммутативности и ассоциативности. function – определенная пользователем функция, которая должна иметь следующие аргументы: invec, inoutvec, len и datatype.

**Освобождение определенного пользователем дескриптора функции**

```
int MPI_Op_free( MPI_Op *op)
```

INOUT	op	операция (дескриптор)
-------	----	--------------------------

Функция MPI\_OP\_FREE маркирует определенную пользователем операцию редукции для удаления и устанавливает значение MPI\_OP\_NULL для аргумента op.

**Выполнение глобальной операции над данными от всех процессов и посылка результата обратно всем процессам**

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

IN	sendbuf	начальный (альтернатива)	адрес	буфера	посылки
OUT	recvbuf	начальный (альтернатива)	адрес	буфера	приема
IN	count	количество посылок (целое)	элементов	в	буфере
IN	datatype	тип данных	элементов	буфера	посылки
IN	op	операция (дескриптор)			
IN	comm	коммуникатор (дескриптор)			

Функция MPI\_ALLREDUCE отличается от MPI\_REDUCE тем, что результат появляется в буфере приема у всех членов группы.

## Выполнение редукции и рассылка результатов

```
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int
*recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

IN	<code>sendbuf</code>	начальный адрес буфера посылки (альтернатива)
OUT	<code>recvbuf</code>	начальный адрес буфера приема (альтернатива)
IN	<code>recvcounts</code>	целочисленный массив, определяющий количество элементов результата, распределенных каждому процессу. Массив должен быть идентичен во всех вызывающих процессах.
IN	<code>datatype</code>	тип данных элементов буфера входа (дескриптор)
IN	<code>op</code>	операция (дескриптор)
IN	<code>comm</code>	коммуникатор (дескриптор)

Функция `MPI_REDUCE_SCATTER` сначала производит поэлементную редукцию вектора из  $count = \sum_i \text{recvcount}[i]$  элементов в буфере посылки, определенном `sendbuf`, `count` и `datatype`. Далее полученный вектор результатов разделяется на  $p$  непересекающихся сегментов, где  $p$  – число членов в группе. Сегмент  $i$  содержит `recvcount[i]` элементов.  $i$ -й сегмент посыпается  $i$ -му процессу и хранится в буфере приема, определяемом `recvbuf`, `recvcounts[i]` и `datatype`.

## Вычисление частичной редукции данных на совокупности процессов

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

IN	<code>sendbuf</code>	начальный адрес буфера посылки (альтернатива)
OUT	<code>recvbuf</code>	начальный адрес буфера приема (альтернатива)
IN	<code>count</code>	количество элементов в буфере приема (целое)
IN	<code>datatype</code>	тип данных элементов в буфере приема (дескриптор)
IN	<code>op</code>	операция (дескриптор)
IN	<code>comm</code>	коммуникатор (дескриптор)

Функция `MPI_SCAN` используется, чтобы выполнить префиксную редукцию данных, распределенных в группе. Операция возвращает в приемный буфер процесса  $i$  редукцию значений в посылающих буферах процессов с номерами  $0, \dots, i$  (включительно). Тип поддерживаемых

операций, их семантика, и ограничения на буфера посылки и приема – такие же, как и для функции MPI\_REDUCE.

### 5. Группы, контексты и коммуникаторы

Определение размера группы

```
int MPI_Group_size(MPI_Group group, int *size)
```

IN	group	группа (дескриптор)
OUT	size	количество процессов в группе (целое)

Определение номера процесса в группе

```
int MPI_Group_rank(MPI_Group group, int *rank)
```

IN	group	группа (дескриптор)
OUT	rank	номер процесса в группе или MPI_UNDEFINED, если процесс не является членом группы (целое)

Перевод номера процесса в одной группе в номер в другой группе

```
int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1, MPI_Group group2, int *ranks2)
```

Сравнение двух групп

```
int MPI_Group_compare (MPI_Group group1, MPI_Group group2, int *result)
```

Доступ к группе, связанной с данным коммуникатором

```
int MPI_Comm_group (MPI_Comm comm, MPI_Group *group)
```

Создание новой группы путем объединения двух групп

```
int MPI_Group_union (MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
```

Создание группы на основе пересечения двух групп

```
int MPI_Group_intersection (MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
```

Создание группы по разности двух групп

```
int MPI_Group_difference (MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
```

Создание группы путем переупорядочивания существующей группы и отбора только тех элементов, которые указаны в списке

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks,  
MPI_Group *newgroup)
```

Создание группы путем переупорядочивания существующей группы и отбора только тех элементов, которые не указаны в списке

```
int MPI_Group_excl(MPI_Group group, int n, int *ranks,  
MPI_Group *newgroup)
```

Создание новой группы из ряда номеров существующей группы

```
int MPI_Group_range_incl(MPI_Group group, int n, int  
ranges[][3], MPI_Group *newgroup)
```

Создание группы путем исключения ряда процессов из существующей группы

```
int MPI_Group_range_excl(MPI_Group group, int n, int  
ranges[][3], MPI_Group *newgroup)
```

Деструктор группы

```
int MPI_Group_free(MPI_Group *group)
```

Определение размера связанный с коммуникатором группы

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Определение номера процесса в коммуникаторе

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Сравнение двух коммуникаторов

```
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int  
*result)
```

Создание нового коммуникатора путем дублирования существующего со всеми его параметрами

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

Создание нового коммуникатора

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm  
*newcomm)
```

Создание нового коммуникатора на основе признаков и ключей

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,  
MPI_Comm *newcomm)
```

Маркирование коммуникатора для удаления

```
int MPI_Comm_free(MPI_Comm *comm)
```

Проверка на соответствие коммуникатора интеркоммуникатору

```
int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
```

Нахождение номера процесса в удаленной группе

```
int MPI_Comm_remote_size(MPI_Comm comm, int *size)
```

Нахождение удаленной группы в коммуникаторе

```
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
```

Создание интеркоммуникатора из двух интеркоммуникаторов

```
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)
```

Создание интеркоммуникатора из интеркоммуникатора

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm *newintracomm)
```

Средства по кэшированию

Генерация нового ключа атрибута

```
int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function *delete_fn, int *keyval, void* extra_state)
```

Освобождение ключа атрибута

```
int MPI_Keyval_free(int *keyval)
```

Хранение связанного с ключом значения атрибута

```
int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)
```

Обрабатывание значения атрибута по ключу

```
int MPI_Attr_get(MPI_Comm comm, int keyval, void* attribute_val, int *flag)
```

Удаление связанного с ключом значения атрибута

```
int MPI_Attr_delete(MPI_Comm comm, int keyval)
```

## *6. Топологии процессов*

Создание нового коммуникатора по заданной топологической информации

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
int *periods, int reorder, MPI_Comm *comm_cart)
```

Распределение процессов по размерностям

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

Создание нового коммуникатора согласно топологической информации

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int
*index, int *edges, int reorder, MPI_Comm *comm_graph)
```

Определение типа связанной с коммуникатором топологии

```
int MPI_Topo_test(MPI_Comm comm, int *status)
```

Получение информации о связанной с коммуникатором топологии

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int
*nedges)
```

Получение информации о связанной с коммуникатором графовой топологии

```
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges,
int *index, int *edges)
```

Получение информации о связанной с коммуникатором картезианской топологии

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

Получение связанной с коммуникатором информации о картезианской топологии

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int
*periods, int *coords)
```

Определение номера процесса в коммуникаторе с картезианской топологией

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

Определение координат в картезианской топологии в соответствии с его номером

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int
*coords)
```

Определение числа соседей узла в графовой топологии

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int
*nneighbors)
```

Определение соседей узла в графовой топологии

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int *neighbors)
```

Определение новых номеров процессов отправителя и получателя после сдвига в заданном направлении

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)
```

Разделение коммуникатора на подгруппы, которые формируют картезианские решетки меньшей размерности

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)
```

Отображение процесса в соответствии с топологической информацией

```
int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods, int *newrank)
```

Размещение процесса согласно информации о топологии графа

```
int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges, int *newrank)
```

## 7. Управление исполнительной средой MPI

Получение номера процесса

```
int MPI_Get_processor_name(char *name, int *resultlen)
```

OUT	name	уникальный фактического (в виртуальному) узла	описатель для противоположность
OUT	resultlen	длина (в печатных результата, возвращаемого в name знаках)	

Эта процедура возвращает имя процессора, на котором был выполнен вызов. Для максимальной гибкости имя является символьной строкой. От этого значения обязательно требуется конкретное описание единицы оборудования, например: «процессор 9 в стойке 4 из mpp.cs.org» и «231» (где 231 – фактический номер процессора в текущей однородной системе).

Аргумент name должен иметь длину не менее MPI\_MAX\_PROCESSOR\_NAME знаков. MPI\_GET\_PROCESSOR\_NAME может добавить к этому много символов в name. Фактическое число символов возвращается в выходном параметре resultlen.

## Обработка ошибок

Создание обработчиков ошибок в стиле MPI

```
int MPI_Errhandler_create(MPI_Handler_function *function,  
MPI_Errhandler *errhandler)
```

Установление обработчика ошибок для коммуникатора

```
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler  
*errhandler)
```

Создание обработчика ошибок для коммуникатора

```
int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler  
*errhandler)
```

Освобождение обработчика ошибок в стиле MPI

```
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
```

Нахождение строки для кода данной ошибки

```
int MPI_Error_string(int errorcode, char *string, int  
*resultlen)
```

Преобразование кода ошибки в класс ошибки

```
int MPI_Error_class(int errorcode, int *errorclass)
```

Таймеры и синхронизация

Нахождение полного времени выполнения операций на используемом процессоре

```
int double MPI_Wtime(void)
```

Функция MPI\_WTIME возвращает количество секунд, представляя полное время по отношению к некоторому моменту времени в прошлом. Этот момент в прошлом не изменяется на протяжении времени жизни процесса. Эта функция универсальна, поскольку она возвращает секунды, а не количество тактовых импульсов.

Нахождение величины разрешения при измерении времени

```
int double MPI_Wtick(void)
```

Функция MPI\_WTICK возвращает величину разрешения MPI\_WTIME в секундах, то есть время в секундах между последовательными импульсами, представленное с двойной точностью. Например, если часы выполнены как аппаратный счетчик, который

инкрементируется каждую миллисекунду, то значение, возвращаемое MPI\_WTICK, должна быть  $10^{-8}$ .

## Подготовка к запуску параллельных программ

### Инициализация параллельных вычислений

```
int MPI_Init(int *argc, char ***argv)
```

Эта процедура должна быть вызвана прежде, чем будет вызвана какая-либо другая MPI подпрограмма (кроме MPI\_INITIALIZED). Она должна быть вызвана всего один раз; последующие вызовы неверны (см. MPI\_INITIALIZED). Все MPI программы должны содержать вызов MPI\_INIT.

### Завершение выполнения программы MPI

```
int MPI_Finalize(void)
```

Эта процедура очищает все состояния MPI. Никакая другая процедура MPI (даже MPI\_INIT) не может быть вызвана после выполнения этой процедуры. Пользователь обязан гарантировать, что все незаконченные обмены будут завершены прежде, чем будет вызвана MPI\_FINALIZE.

### Определение выполнения MPI\_Init

```
MPI_Initialized(int *flag)
```

OUT flag параметр flag принимает значение true, если MPI\_INIT была вызвана и false в противном случае.

Процедура MPI\_INITIALIZED может использоваться для определения того, был ли вызван MPI\_INIT. Это единственная процедура, которая может быть вызвана перед MPI\_INIT.

### Прекращение выполнения операций

```
MPI_Abort(MPI_Comm comm, int errorcode)
```

IN comm коммуникатор прерываемой задачи  
IN errorcode код ошибки для возврата в среду исполнения

Процедура MPI\_ABORT пробует наилучшим способом прервать выполнение всех задач в группе коммуникатора comm. Эта процедура не требует, чтобы исполнительная среда производила какие-либо действия с

кодом ошибки. Однако среда Unix должна обрабатывать его как return errorcode из главной программы или abort (errorcode).

## 8. Нерекомендуемые имена и функции стандарта MPI1.1

Нерекомендуемые	Замена MPI-2
MPI ADDRESS	MPI GET ADDRESS
MPI TYPE HINDEXED	MPI TYPE CREATE_HINDEXED
MPI TYPE HVECTOR	MPI TYPE CREATE_HVECTOR
MPI TYPE STRUCT	MPI TYPE CREATE_STRUCT
MPI TYPE EXTENT	MPI TYPE GET_EXTENT
MPI TYPE UB	MPI TYPE GET_EXTENT
MPI TYPE LB	MPI TYPE GET_EXTENT
MPI LB	MPI TYPE CREATE_RESIZED
MPI UB	MPI TYPE CREATE_RESIZED
MPI_ERRHANDLER_CREATE	MPI COMM CREATE_ERRHANDLER
MPI_ERRHANDLER_GET	MPI COMM GET_ERRHANDLER
MPI_ERRHANDLER_SET	MPI COMM SET_ERRHANDLER
MPI Handler function	MPI Comm_errhandler_fn
MPI_KEYVAL_CREATE	MPI COMM CREATE_KEYVAL
MPI_KEYVAL_FREE	MPI COMM FREE_KEYVAL
MPI_DUP_FN	MPI COMM DUP_FN
MPI_NULL_COPY_FN	MPI COMM NULL_COPY_FN
MPI_NULL_DELETE_FN	MPI COMM NULL_DELETE_FN
MPI_Copy function	MPI Comm_copy_attr_function
COPY FUNCTION	COMM COPY_ATTR_FN
MPI_Delete function	MPI Comm_delete_attr_function
DELETE FUNCTION	COMM_DELETE_ATTR_FN
MPI_ATTR_DELETE	MPI COMM DELETE_ATTR
MPI_ATTR_GET	MPI COMM GET_ATTR
MPI_ATTR_PUT	MPI COMM SET_ATTR

## Исходные тексты программ прототипов

В данном приложении приведены тексты программ на языке С с использованием библиотеки MPI для прототипов объединения подпрограмм по логической функции «И», «ИЛИ», «И-ИЛИ». Дополнительно представлен обработчик выходных данных прототипов на языке Perl и тексты программ с использованием пакета MathCad.

### 1. Исходные тексты программ на языке С с использованием библиотеки MPI

#### 1.1. Прототип объединения подпрограмм по логической функции «И»

Основной модуль: nsCall\_v1.cpp

```

/*
 * file:    nsCall.cpp
 * purpose: General process functions
 */

// *****
// includes
#define nsCALL_V1
#define nsIMM_PRINTF_ONLY
#include <direct.h>
#include <string.h>
#include "nsMain.h"
#include "nsConfig.h"
#include "nsProb.h"
#include "nsCall.h"

// *****
// defines
// optional defines, comment any 'n'usable
#define nsCALL_NO_INPUT
//#define nsCALL_MATRIX_DBG

// default matrix name
#ifndef nsCALL_NO_INPUT
#define nsCALL_DEF_MATRIXNAME "test"
#endif

// *****
// types
typedef struct tagNsCallLocalData
{
    nsCallState est;
    int nMRows, nMcols;
    NSreal rDelta;
} NsCallLocalData;

typedef struct tagNsMatrixData
{
    Matrix *mpMatrix;
} NSMatrixData;

// *****
// functions
// *****
* notes: only process with ID == 0 must call me!
*/
NsCallState NsCallReadMatrices(NSMatrixData *mdaMatrices, int nMatCount, char *cpFN)
{
    char sFileName[35];

```

```

NsCallState eRetState = nsSTATE_NA;
int ni;
char sFileNameIdx[45];
char sFullFN[256];

#ifndef nsDEBUG_MSG
int nCurrid;
MPI_Comm_rank(MPI_COMM_WORLD, &nCurrid);
nsASSERT(nCurrid == 0);
#endif

if (cpFN == NULL || strlen(cpFN) == 0)
{
    nsPRINTF_I(((" Введите имя файла содержащего матрицу (*.nsCALL_MATRIXFILEEXT"), or 'q' for
quit\n"));
    scanf("%30s", sFileName);
}
else
    strcpy(sFileName, cpFN);

// keyword 'q' means EXIT
if (sFileName[0] != 'q')
{
    for (ni = 0; ni < nMatCount; ni++)
    {
        // init
        mdaMatrices[ni].mpMatrix = NULL;

        sprintf(sFileNameIdx, "%d", sFileName, ni);

        // add file extension if needed
        if (strchr(sFileNameIdx, '.') == NULL)
            strcat(sFileNameIdx, ".nsCALL_MATRIXFILEEXT");

        // #00HACK test file
        {
            getcwd(sFullFN, sizeof(sFullFN));
            strcat(sFullFN, "\\");
            strcat(sFullFN, sFileNameIdx);

            FILE *fp = fopen(sFullFN, "rt");
            if (fp == NULL)
            {
                sprintf(sFileNameIdx, "%s", sFileName);

                if (strchr(sFileNameIdx, '.') == NULL)
                    strcat(sFileNameIdx, ".nsCALL_MATRIXFILEEXT");

                getcwd(sFullFN, sizeof(sFullFN));
                strcat(sFullFN, "\\");
                strcat(sFullFN, sFileNameIdx);
            }
            else
                fclose(fp);
        }

#ifndef nsCALL_MATRIX_DBG
        nsPRINTF_DBG(("::nsCallReadMatrices: Main process trying to load: %s\n", sFullFN));
#endif
    }
}

try
{
    // try to load matrix
    Matrix mTmp(sFullFN);

    // check matrix & decide what to do
    if (NSprobMatrixIsGood(&mTmp) == TRUE)
    {
        // create matrix
        mdaMatrices[ni].mpMatrix = new Matrix(mTmp);

#ifndef nsCALL_MATRIX_DBG
        nsPRINTF_DBG(("::nsCallReadMatrices: mtr # %d\n", ni));
        mdaMatrices[ni].mpMatrix->print();
#endif
    }
    else
        eRetState = nsSTATE_START;
}
else
{
    nsPRINTF(("Matrix \\"%s\\" don't obey conditions. Program stopped...\n", sFileNameIdx));
    eRetState = nsSTATE_EXIT;
    break;
}

catch(Matrix::exception e)
{
    nsPRINTF(("::nsCallReadMatrices:Matrix EXCEPTION, code: %i\n", e.ec()));
    eRetState = nsSTATE_EXIT;
    break;
}

else
    eRetState = nsSTATE_EXIT;

return eRetState;
}

```

```

void NsCallUnReadMatricesAndDestroyStacks(NsMatrixData *mdaMatrices, int nMatCount, NsStackData *sdpArray,
                                         int nNumPrc)
{
    int nI;
    for (nI = 0; nI < nMatCount; nI++)
    {
        if (mdaMatrices[nI].mpMatrix != NULL)
        {
            delete mdaMatrices[nI].mpMatrix;
            mdaMatrices[nI].mpMatrix = NULL;
        }
    }

    for (nI = 0; nI < nNumPrc; nI++)
    {
        if (sdpArray[nI].spS != NULL)
        {
            delete sdpArray[nI].spS;
            sdpArray[nI].spS = NULL;
        }
    }
}

// -----
// notes: Only process with ID == 0 must call me!
void NsCallMakeStepONE(Matrix &mQ, NsStack &sFirstF, NsCallLocalData *spData)
{
    int nCurrid;
    MPI_Comm_rank(MPI_COMM_WORLD, &nCurrid);
    nsassert(nCurrid == 0);

    // calculate distribution
    NsProbCalcDistribution(mQ, spData->rDelta, sFirstF);

    // out values
    nsprintf(" Плотность распределения вер-ти времени выполнения (процесс %d):\n ", nCurrid+1);
    #ifndef NSIMM_PRINTF_ONLY
    sFirstF.Print();
    #endif
    nsprintf(" Параметр m = %d (дельта = %.10lf).\n\n", sFirstF.LastIdx(), spData->rDelta);
}

// -----
void NsCallMakeStepTWO(Matrix &mQ2, NsStack &sFirst, NsStack &sSecond, int nMyID, NsCallLocalData *spData)
{
    NsStack sTmp(20);
    // calculate distrib
    NsProbCalcDistribution(mQ2, spData->rDelta, sTmp);

    nsprintf(" Плотность распределения вер-ти времени выполнения (процесс %d):\n ", nMyID+1);
    #ifndef NSIMM_PRINTF_ONLY
    sTmp.Print();
    #endif
    nsprintf(" Параметр m = %d (дельта = %.10lf).\n", sTmp.LastIdx(), spData->rDelta);

    // make convolution
    NsProbCalcConvolution(sFirst, sTmp, sSecond);

    // out values
    nsprintf(" Свертка (процесс %d):\n ", nMyID+1);
    #ifndef NSIMM_PRINTF_ONLY
    sSecond.Print();
    #endif
    nsprintf(" Параметр m = %d (дельта = %.10lf).\n\n", sSecond.LastIdx(), spData->rDelta);
}

// -----
BOOL NsCallMain(int nNumPrc, int nMyID, char *cpfn)
{
    int nI, nFirstSize;
    Matrix *mpForTWO;

    // distribution stacks
    NsStack SFT(20), SFT_m(20), SFTN(20);

    NsStack STimes(20);

    int nMatrixCount = nNumPrc + 2;

    NsCallLocalData *spData;
    NsMatrixData *spMtrData;
    NsStackData *spStkData;

    // master process: prepare data
    if (nMyID == 0)
    {
        spMtrData = new NsMatrixData[nMatrixCount];
        spData = new NsCallLocalData[nMatrixCount];
    }

    // read matrix(es) from file(s)
}

```

```

spData[0].est = NsCallReadMatrices(spMtrData, nMatrixCount, cpFN);
if (spData[0].est != nsSTATE_START)
{
    nsPRINTF_INM(("::NsCallMain: User requested exit...\n"));
    for (NI = 1; NI < nNumPrc; NI++)
        MPI_Send(spData, sizeof(NsCallLocalData), MPI_BYTE, NI, nsMSGTAG_DATA, MPI_COMM_WORLD);
    delete [] spMtrData;
    delete [] spData;
    return TRUE;
}

// store data for all process
for (NI = 0; NI < nMatrixCount; NI++)
{
    spData[NI].est = spData[0].est;
    spData[NI].nMacols = spMtrData[NI].mpMatrix->cols();
    spData[NI].nMarows = spMtrData[NI].mpMatrix->rows();
    spData[NI].rDelta = Nsconfigget()->rDelta;
}

// broadcast state & matrix
// m0, (m1, m2, ..., m[nNumPrc]), m[nNumPrc+1]
for (NI = 1; NI < nNumPrc; NI++)
{
    MPI_Send(&spData[NI-1], sizeof(NsCallLocalData), MPI_BYTE, NI, nsMSGTAG_DATA, MPI_COMM_WORLD);
    MPI_Send(spMtrData[NI].mpMatrix->data, spMtrData[NI].mpMatrix->cols() *
        spMtrData[NI].mpMatrix->rows() *
        sizeof(tMatrixElement), MPI_BYTE, NI, nsMSGTAG_MATRIX_ENTRY, MPI_COMM_WORLD);
}
}

else
{
    spMtrData = new NsMatrixData[1];
    spData = new NsCallLocalData;
    MPI_Status sStat;

    // broadcast state
    MPI_Recv(spData, sizeof(NsCallLocalData), MPI_BYTE, 0, nsMSGTAG_DATA, MPI_COMM_WORLD, &sStat);

    if (spData->est != nsSTATE_START)
    {
        delete [] spMtrData;
        delete spData;
        return TRUE;
    }

    // Create matrix
    spMtrData[0].mpMatrix = new Matrix(spData->nMarows, spData->nMacols);

    MPI_Recv(spMtrData[0].mpMatrix->data, spMtrData[0].mpMatrix->cols() * spMtrData[0].mpMatrix->rows() *
        sizeof(tMatrixElement), MPI_BYTE, 0, nsMSGTAG_MATRIX_ENTRY, MPI_COMM_WORLD, &sStat);
}

sTimes += MPI_Wtime();

// ONE #####
if (nMyID == 0)
{
    nsPRINTF(("##### Tran M1 #####\n"));
    // first step here
    NsCallMakeStepONE(*spMtrData[0].mpMatrix), SFT1, &spData[0]);

    sTimes += MPI_Wtime();
    // send first distro to all
    nFirstSize = SFT1.LastIdx() + 1;
    MPI_Bcast(nFirstSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(sFT1.pData, nFirstSize * sizeof(NsReal), MPI_BYTE, 0, MPI_COMM_WORLD);
}

else
{
    sTimes += MPI_Wtime();
    // receive first distribution size
    MPI_Bcast(nFirstSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
    // reallocate buffer
    SFT1.SetLastIdx(nFirstSize - 1);
    // receive distribution
    MPI_Bcast(sFT1.pData, nFirstSize * sizeof(NsReal), MPI_BYTE, 0, MPI_COMM_WORLD);
}

sTimes += MPI_Wtime();

// TWO #####
if (nMyID == 0)
{
    mpForTwo = spMtrData[1].mpMatrix;
    spStkData = new NsStackData[nNumPrc];
}

else
{
    mpForTwo = spMtrData[0].mpMatrix;
    spStkData = new NsStackData[1];
}

spStkdata[0].spS = new NsStack(20);

if (nMyID == 0)

```

```

{nsprintf(("##### Этап №2 #####\n"));
NsCallMakeStepTwo(*mpForTwo, sFT1, *(spStkData[0].spS), nMyID, spData);
sTimes += MPI_Wtime();
if (nMyID == 0)
{
    int nSize;
    MPI_Status sStat;
    // receive all distributions
    for (nI = 1; nI < nNumPrc; nI++)
    {
        MPI_Recv(&nSize, 1, MPI_INT, nI, nMSGTAG_SLAVE_DISTSIZE, MPI_COMM_WORLD, &sStat);
        spStkData[nI].spS = new NsStack(nSize);
        MPI_Recv(spStkData[nI].spS->pData, sizeof(NsReal) * nSize, MPI_BYTE, nI, nMSGTAG_SLAVE_DISTENTRY,
        MPI_COMM_WORLD, &sStat);
        spStkData[nI].spS->SetLastIdx(nSize - 1);
    }
}
else
{
    int nSize = spStkData[0].spS->LastIdx() + 1;
    MPI_Send(&(nSize), 1, MPI_INT, 0, nMSGTAG_SLAVE_DISTSIZE, MPI_COMM_WORLD);
    MPI_Send(spStkData[0].spS->pData, sizeof(NsReal) * nSize, MPI_BYTE, 0, nMSGTAG_SLAVE_DISTENTRY,
    MPI_COMM_WORLD);
}
sTimes += MPI_Wtime();
// THREE #####
if (nMyID == 0)
{
    NsReal rAvg;
    NsReal rDisp;

    NsProbCalcANDUnion(spStkData, nNumPrc, sFT_m);
    nsprintf(("##### Этап №3 #####\n AND результат:\n "));
#ifndef NSIMM_PRINTF_ONLY
    sFT_m.Print();
#endif
}
sTimes += MPI_Wtime();
NsCallMakeStepTwo(*(spMtrData[nNumPrc+1].mpMatrix), sFT_m, sFTN, nMyID, &spData[nNumPrc+1]);
// calculate M & D
rAvg = sFTN.GetAverage();
rDisp = sFTN.GetDispersion(rAvg);

sTimes += MPI_Wtime();
nsprintf_lnn("Среднее = %.4lf, дисперсия = %.4lf\n##### Вычисления завершены ##### (cols = %d)\n",
rAvg, rDisp, spMtrData[0].mpMatrix->cols));
}
else
{
    sTimes += MPI_Wtime();
}

// timing statistic
MPI_Barrier(MPI_COMM_WORLD);
nsprintf_lnn(("Процесс %d, Дельта: %.8lf, тайминг: %f", nMyID, sTimes[sTimes.LastIdx()]-sTimes[0]));
sTimes.Print();
nsprintf_lnn(("Коммуникации суммарно: %.8lf, вычисления суммарно: %.8lf\n\n", sTimes[2]-sTimes[1] +
sTimes[4]-sTimes[3], sTimes[1]-sTimes[0] + sTimes[3]-sTimes[2] + sTimes[5]-sTimes[4]));

// destroy objects
if (nMyID == 0)
{
    NsCallUnReadMatricesAndDestroyStacks(spmtrData, nMatrixCount, spStkData, nNumPrc);
    delete [] spMtrData;
    delete [] spData;
}
else
{
    NsCallUnReadMatricesAndDestroyStacks(spmtrData, 1, spStkData, 1);
    delete [] spMtrData;
    delete spData;
}

#ifndef NSCALL_NO_INPUT
return FALSE;
#else
return TRUE;
#endif
}

```

## 1.2. Прототип объединения подпрограмм по логической функции «ИЛИ»

Основной модуль: nsCall\_v2.cpp

/\* file: nsCall.cpp

```

* purpose: General process functions
*/
// **** includes ****
#define nsCALL_V2
#define nsIMML_PRINTF_ONLY
#include <direct.h>
#include <string.h>
#include "nsMain.h"
#include "nsConfig.h"
#include "nsProb.h"
#include "nsCall.h"

// **** defines ****
// optional defines, comment any n'usable
#define nsCALL_NO_INPUT
#define nsCALL_MATRIX_DBG

// default matrix name
#define nsCALL_NO_INPUT
#define nsCALL_DEF_MATRIXNAME "test"
#endif

// **** types ****
typedef struct tagNsCallLocalData
{
    NsCallState est;
    int rmaxRows, rmaxCols;
    NsReal rDelta;
} NsCallLocalData;

typedef struct tagNsMatrixData
{
    Matrix *mpMatrix;
} NsMatrixData;

// **** functions ****
// -----
/*
 * notes: Only process with ID == 0 must call me!
 */
NsCallState NsCallReadMatrices(NsMatrixData *mdaMatrices, int nMatCount, char *cpFN)
{
    char sFileName[35];
    NsCallState eketState = nsSTATE_NA;
    int ni;
    char sFileNameIdx[45];
    char sFullFN[256];

#ifndef nsDEBUG_MSG
    int nCurrid;
    MPI_Comm_rank(MPI_COMM_WORLD, &nCurrid);
    nsASSERT(nCurrid == 0);
#endif

    if (cpFN == NULL || strlen(cpFN) == 0)
    {
        nsPRINTF_IMM(" Введите имя файла содержащего матрицу (*.\"nsCALL_MATRIXFILEEXT\"), or 'q' for quit(\"q\")");
        scanf("%30s", sFileName);
    }
    else
        strcpy(sFileName, cpFN);

    // keyword 'q' means EXIT
    if (sFileName[0] != 'q')
    {
        for (ni = 0; ni < nMatCount; ni++)
        {
            // init
            mdaMatrices[ni].mpMatrix = NULL;
            sprintf(sFileNameIdx, "%s%d", sFileName, ni);

            // add file extension if needed
            if (strchr(sFileNameIdx, '.') == NULL)
                strcat(sFileNameIdx, ".\"nsCALL_MATRIXFILEEXT\"");

            // ***HACK test file
            getcwd(sFullFN, sizeof(sFullFN));
            strcat(sFullFN, "\\\"");
        }
    }
}

```

```

strcat(sFullFN, sFileNameIdx);
FILE *fp = fopen(sFullFN, "rt");
if (fp == NULL)
{
    sprintf(sFileNameIdx, "%s", sFileName);
    if ( strchr(sFileNameIdx, '.') == NULL)
        strcat(sFileNameIdx, ".nsCALL_MATRIXFILEEXT");
    getcwd(sFullFN, sizeof(sFullFN));
    strcat(sFullFN, "\\");
    strcat(sFullFN, sFileNameIdx);
}
else
fclose(fp);

#endif nsCALL_MATRIX_DBG
nsPRINTE_DBG(("::NsCallReadMatrices: Main process trying to load: %s\n", sFullFN));
#endif
try
{
    // try to load matrix
    Matrix mtmp(sFullFN);

    // check matrix & decide what to do
    if (NsProMatrixIsGood(&tmp) == TRUE)
    {
        // create matrix
        mdaMatrices[nI].mpMatrix = new Matrix(mtmp);
#ifndef nsCALL_MATRIX_DBG
        nsPRINTE_DBG(("::NsCallReadMatrices: mtr # %d\n", nI));
        mdaMatrices[nI].mpMatrix->print();
#endif
        eRetState = nsSTATE_START;
    }
    else
    {
        nsPRINTE(("Matrix \"%s\" don't obey conditions. Program stopped...\n", sFileNameIdx));
        eRetState = nsSTATE_EXIT;
        break;
    }
}
catch(Matrix::exception e)
{
    nsPRINTE(("::NsCallReadMatrices:Matrix EXCEPTION, code: %i\n", e.ec()));
    eRetState = nsSTATE_EXIT;
    break;
}
}
else
eRetState = nsSTATE_EXIT;
return eRetState;
}

void NsCallUnReadMatricesAndDestroyStacks(NsMatrixData *mdaMatrices, int nMatCount, NsStackData *sdpArray,
int nNumPrc)
{
    int nI;
    for (nI = 0; nI < nMatCount; nI++)
    {
        if (mdaMatrices[nI].mpMatrix != NULL)
        {
            delete mdaMatrices[nI].mpMatrix;
            mdaMatrices[nI].mpMatrix = NULL;
        }
    }

    for (nI = 0; nI < nNumPrc; nI++)
    {
        if (sdpArray[nI].sps != NULL)
        {
            delete sdpArray[nI].sps;
            sdpArray[nI].sps = NULL;
        }
    }
}

// =====
/* notes: Only process with ID == 0 must call me!
void NsCallMakeStepONE(Matrix &M, NsStack &FirstF, NsCallLocalData *spData)
{
    int nCurrid;
    MPI_Comm_rank(MPI_COMM_WORLD, &nCurrid);
    NSASSERT(nCurrid == 0);

    // calculate distribution
    NsProbCalcDistribution(M, spData->rDelta, sFirstF);

    // out values
    nsPRINTE(("плотность распределения вер-ти времени выполнения (процесс %d):\n ", nCurrid+1));
}

```

```

#ifndef nsIMM_PRINTF_ONLY
    sFirstF.Print();
#endif
NSPRINT((" Параметр m = %d (дельта = %.10f).\n", sFirstF.LastIdx(), spData->rDelta));
}

// -----
void NsCallMakeStepTwo(Matrix &mQ2, NsStack &sFirst, NsStack &sSecond, int nMyID, NsCallLocalData *spData)
{
    NsStack stMP(20);

    // calculate distrib
    NsProbCalcDistribution(mQ2, spData->rDelta, stMP);

    NSPRINT((" Плотность распределения вер-ти времени выполнения (процесс %d):\n ", nMyID+1));
    #ifndef nsIMM_PRINTF_ONLY
        stMP.Print();
    #endif
    NSPRINT((" Параметр m = %d (дельта = %.10f).\n", stMP.LastIdx(), spData->rDelta));

    // make convolution
    NsProbCalcConvolution(sFirst, stMP, sSecond);

    // out values
    NSPRINT((" Смертка (процесс %d):\n ", nMyID+1));
    #ifndef nsIMM_PRINTF_ONLY
        sSecond.Print();
    #endif
    NSPRINT((" Параметр m = %d (дельта = %.10f).\n", sSecond.LastIdx(), spData->rDelta));
}

// -----
BOOL NsCallMain(int nNumPrc, int nMyID, char *cpFN)
{
    int      nI, nFirstSize;
    Matrix  *mpForTwo_First;
    Matrix  *mpForTwo_Second;

    // Distribution stacks
    NsStack  SFTI(20), SFT_M(20), SFTN(20);

    NsStack  STimes(20);

    int      nMatrixCount = nNumPrc * 2 + 2;

    NsCallLocalData *spData;
    NsMatrixData  *spMtrData;
    NsStackData   *spStkData;

    // master process: prepare data
    if (nMyID == 0)
    {
        spMtrData  = new NsMatrixData[nMatrixCount];
        spData     = new NsCallLocalData[nMatrixCount];

        // read matrix(es) from file(s)
        spData[0].est = NsCallReadMatrices(spMtrData, nMatrixCount, cpFN);
        if (spData[0].est != nsSTATE_START)
        {
            NSPRINTF_IMM(("::NsCallMain: User requested exit...\n"));

            for (nI = 1; nI < nNumPrc; nI++)
                MPI_Send(&spData[nI].est, sizeof(NsCallLocalData), MPI_BYTE, nI, nsMSGTAG_DATA_FIRST_SECOND,
                        MPI_COMM_WORLD);

            delete [] spMtrData;
            delete [] spData;
            return TRUE;
        }

        // store data for all process
        for (nI = 0; nI < nMatrixCount; nI++)
        {
            spData[nI].est     = spData[0].est;
            spData[nI].nMacols = spMtrData[nI].mpMatrix->cols();
            spData[nI].nMrows  = spMtrData[nI].mpMatrix->rows();
            spData[nI].rDelta  = NsConfigGet()->rDelta;
        }

        // broadcast state & matrix
        // n0, (m1, m2, ..., M(nNumPrc*2)),  m(nNumPrc*2+1)
        for (nI = 1; nI < nNumPrc; nI++)
        {
            MPI_Send(&(spData[nI*2+1]), sizeof(NsCallLocalData)*2, MPI_BYTE, nI, nsMSGTAG_DATA_FIRST_SECOND,
                    MPI_COMM_WORLD);

            MPI_Send(spMtrData[nI+1].mpMatrix->data, spMtrData[nI+1].mpMatrix->cols() *
                sizeof(tMatrixElement), MPI_BYTE, nI, nsMSGTAG_MATRIX_ENTRY_FIRST, MPI_COMM_WORLD);
            MPI_Send(spMtrData[nI+1].mpMatrix->data, spMtrData[nI+1].mpMatrix->cols() *
                sizeof(tMatrixElement), MPI_BYTE, nI, nsMSGTAG_MATRIX_ENTRY_SECOND, MPI_COMM_WORLD);
        }
    }
}

```

```

spMtrData     = new NsMatrixData[2];
spData       = new NsCallLocalData[2];
MPI_Status    sStat;

// broadcast state
MPI_Recv(spData, sizeof(NsCallLocalData)*2, MPI_BYTE, 0, nsMSGTAG_DATA_FIRST_SECOND, MPI_COMM_WORLD,
&sStat);

if (spData->eSt != nsSTATE_START)
{
    delete [] spMtrData;
    delete [] spData;
    return TRUE;
}

// Create matrix
spMtrData[0].mpMatrix = new Matrix(spData[0].nMaRows, spData[0].nMaCols);
spMtrData[1].mpMatrix = new Matrix(spData[1].nMaRows, spData[1].nMaCols);

MPI_Recv(spMtrData[0].mpMatrix->data, spMtrData[0].mpMatrix->cols() * spMtrData[0].mpMatrix->rows(), MPI_BYTE, 0, nsMSGTAG_MATRIX_ENTRY_FIRST, MPI_COMM_WORLD, &sStat);
MPI_Recv(spMtrData[1].mpMatrix->data, spMtrData[1].mpMatrix->cols() * spMtrData[1].mpMatrix->rows(), MPI_BYTE, 0, nsMSGTAG_MATRIX_ENTRY_SECOND, MPI_COMM_WORLD, &sStat);
}

sTimes += MPI_Wtime();

// ONE #####
if (nMyID == 0)
{
    if (defined(nsCALL_MATRIX_DBG) && defined(nsDEBUG_MSG))
        nsPRINTF("!!! Step 1, MyID = %d, Matrixx1: \n", nMyID);
    spMtrData[0].mpMatrix->print();
}

#endif

nsPRINTF("##### Tran M1 #####\n");
// First step here
NsCallMakeStepONEC(spMtrData[0].mpMatrix), sFT1, &spData[0]);
sTimes += MPI_Wtime();

// send first distro to all
nFirstSize = sFT1.LastIdx() + 1;
MPI_Bcast(&nFirstSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(sFT1.pData, nFirstSize + sizeof(NsReal), MPI_BYTE, 0, MPI_COMM_WORLD);
}
else
{
    sTimes += MPI_Wtime();

    // receive first distribution size
    MPI_Bcast(&nFirstSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
    // reallocate buffer
    sFT1.SetLastIdx(nFirstSize - 1);
    // receive distribution
    MPI_Bcast(sFT1.pData, nFirstSize * sizeof(NsReal), MPI_BYTE, 0, MPI_COMM_WORLD);
}

sTimes += MPI_Wtime();

// TWO #####
if (nMyID == 0)
{
    mpForTwo_First = spMtrData[1].mpMatrix;
    mpForTwo_Second = spMtrData[2].mpMatrix;
    spStkData = new NsStackData[nNumPrc*2];
}
else
{
    mpForTwo_First = spMtrData[0].mpMatrix;
    mpForTwo_Second = spMtrData[1].mpMatrix;
    spStkData = new NsStackData[2];
}

if (defined(nsCALL_MATRIX_DBG) && defined(nsDEBUG_MSG))
    nsPRINTF("!!! Step 2, MyID = %d, Matrixx1: \n", nMyID);
    mpForTwo_First->print();
    nsPRINTF("!!! Step 2, MyID = %d, Matrixx2: \n", nMyID);
    mpForTwo_Second->print();
}

spStkData[0].spS = new NsStack(20);
spStkData[1].spS = new NsStack(20);

sTimes += MPI_Wtime();

if (nMyID == 0)
{
    nsPRINTF("##### Tran M2 #####\n");
    NsCallMakeStepTWO(*mpForTwo_First, sFT1, *(spStkData[0].spS), nMyID, spData);
    NsCallMakeStepTWO(*mpForTwo_Second, sFT1, *(spStkData[1].spS), nMyID, spData);
}

sTimes += MPI_Wtime();

if (nMyID == 0)
{
    int npSize[2];
    MPI_Status sStat;

    // receive all distributions
}
```

```

for (nI = 1; nI < nNumPrc; nI++)
{
    MPI_Recv(&npSize, 2, MPI_INT, nI, nsMSGTAG_SLAVE_DISTSIZE_FIRST_SECOND, MPI_COMM_WORLD, &sStat);
    spstkdata[nI*2+0].sps = new NSStack(npSize[0]);
    spstkdata[nI*2+1].sps = new NSStack(npSize[1]);
    MPI_Recv(&spstkdata[nI*2+0].sps->pData, sizeof(nsReal) * npSize[0], MPI_BYTE, nI,
    nsMSGTAG_SLAVE_DISTENTRY_FIRST, MPI_COMM_WORLD, &sStat);
    spstkdata[nI*2+0].sps->setLastIdx(npSize[0] - 1);
    MPI_Recv(&spstkdata[nI*2+1].sps->pData, sizeof(nsReal) * npSize[1], MPI_BYTE, nI,
    nsMSGTAG_SLAVE_DISTENTRY_SECOND, MPI_COMM_WORLD, &sStat);
    spstkdata[nI*2+1].sps->setLastIdx(npSize[1] - 1);
}

else
{
    int naSize[2];
    naSize[0] = spstkdata[0].sps->LastIdx() + 1;
    naSize[1] = spstkdata[1].sps->LastIdx() + 1;

    MPI_Send(naSize, 2, MPI_INT, 0, nsMSGTAG_SLAVE_DISTSIZE_FIRST_SECOND, MPI_COMM_WORLD);
    MPI_Send(&spstkdata[0].sps->pData, sizeof(nsReal) * naSize[0], MPI_BYTE, 0,
    nsMSGTAG_SLAVE_DISTENTRY_FIRST, MPI_COMM_WORLD);
    MPI_Send(&spstkdata[1].sps->pData, sizeof(nsReal) * naSize[0], MPI_BYTE, 0,
    nsMSGTAG_SLAVE_DISTENTRY_SECOND, MPI_COMM_WORLD);
}

sTimes += MPI_Wtime();

// THREE #####
if (nMyID == 0)
{
    NSReal rAvg;
    NSReal rDisp;

    NSprobCalcANDUnion(&spstkdata, nNumPrc*2, SFT_M);

    // nsPRINTF("##### Этап №3 #####\n AND результат:\n ");
    #ifndef nsIMM_PRINTE_ONLY
    SFT_M.Print();
    #endif

    #if defined(nsCALC_MATRIX_DBG) || defined(nsDEBUG_MSG)
    nsPRINTF_DBG("!! Step 3, MyID = %d Matrix: \n", nMyID);
    spmtrData[nNumPrc*2+1].mpMatrix->print();
    #endif

    NsCallMakestepTWO(*(&spmtrData[nNumPrc*2+1].mpMatrix), SFT_M, SFTN, nMyID, &spData[nNumPrc*2+1]);
    // calculate M & D
    rAvg = SFTN.GetAverage();
    rDisp = SFTN.GetDispersion(rAvg);

    sTimes += MPI_Wtime();
    nsPRINTF_IWM("( Среднее = %.4lf, дисперсия = %.4lf\n##### Вычисления завершены ##### (cols = %d)\n",
    rAvg, rDisp, spmtrData[0].mpMatrix->cols()));
}
else
{
    sTimes += MPI_Wtime();
}

// timing statistic
MPI_BARRIER(MPI_COMM_WORLD);
nsPRINTF_IWM("(Процесс %d, дельта: %.8lf, тайминг: \n", nMyID, sTimes[sTimes.LastIdx()]-sTimes[0]));
sTimes.Print();
nsPRINTF_IWM("(коммуникации суммарно: %.8lf, вычисления суммарно: %.8lf\n\n", sTimes[2]-sTimes[1] +
sTimes[4]-sTimes[3], sTimes[1]-sTimes[0] + sTimes[3]-sTimes[2] + sTimes[5]-sTimes[4]));

// destroy objects
if (nMyID == 0)
    NsCallUnReadMatricesAndDestroyStacks(spmtrData, nMatrixCount, spstkdata, nNumPrc);
else
    NsCallUnReadMatricesAndDestroyStacks(spmtrData, 2, spstkdata, 2);

delete [] spmtrData;
delete [] spData;

#ifndef nsCALL_NO_INPUT
return FALSE;
#else
return TRUE;
#endif
}

```

### 1.3. Прототип объединения подпрограмм по логической функции «И-ИЛИ»

Основной модуль: nsCall\_v3.cpp

```
/*
 * file: nsCall.cpp
 * purpose: General process functions

```

```

*/
// **** includes ****
#define nsCALL_V3
#define nsIMM_PRINTF_ONLY
#include <direct.h>
#include <string.h>
#include "nsMain.h"
#include "nsConfig.h"
#include "nsProb.h"
#include "nsCall.h"

// **** defines ****
// optional defines, comment any n'usable
//#define nsCALL_NO_INPUT
//#define nsCALL_MATRIX_DBG

// default matrix name
#ifndef nsCALL_NO_INPUT
#define nsCALL_DEF_MATRIXNAME "test"
#endif

// **** types ****
typedef struct tagNsCallLocalData
{
    NsCallState est;
    int nRows, nCols;
    NsReal rDelta;
} NsCallLocalData;

typedef struct tagNsMatrixData
{
    Matrix *mpMatrix;
} NsMatrixData;

// **** functions ****
// =====
/*
 * notes: only process with ID == 0 must call me!
 */
NscallState NsCallReadMatrices(NsMatrixData *mdMatrices, int nMatCount, char *cpFN)
{
    char sFileName[35];
    NscallState eRetState = nsSTATE_NA;
    int ni;
    char sFileNameIdx[45];
    char sFullFN[256];

#ifdef nsDEBUG_MSG
    int nCurrid;
    MPI_Comm_rank(MPI_COMM_WORLD, &nCurrid);
    nsASSERT(nCurrid == 0);
#endif

    if (cpFN == NULL || strlen(cpFN) == 0)
    {
        nsPRINTF_IMN(" Введите имя файла содержащего матрицу (*.nsCALL_MATRIXFILEEXT), or 'q' for quit\n");
        scanf("%30s", sFileName);
    }
    else
        strcpy(sFileName, cpFN);

    // keyword 'q' means EXIT
    if (sFileName[0] != 'q')
    {
        for (ni = 0; ni < nMatCount; ni++)
        {
            // init
            mdMatrices[ni].mpMatrix = NULL;
            sprintf(sFileNameIdx, "%s%d", sFileName, ni);

            // add file extension if needed
            if (strchr(sFileNameIdx, '.') == NULL)
                strcat(sFileNameIdx, ".nsCALL_MATRIXFILEEXT");

            // ***HACK test file
            getcwd(sFullFN, sizeof(sFullFN));
            strcat(sFullFN, "\\");
            strcat(sFullFN, sFileNameIdx);
        }
    }
}

```

```

FILE *fp = fopen(sFullFN, "rt");
if (fp == NULL)
{
    sprintf(sFileNameIdx, "%s", sFileName);
    if (strchr(sFileNameIdx, '.') == NULL)
        strcat(sFileNameIdx, ".nsCALL_MATRIXFILEEXT");
    getcwd(sFullFN, sizeof(sFullFN));
    strcat(sFullFN, "\\");
    strcat(sFullFN, sFileNameIdx);
}
else
    fclose(fp);
#endif NSCALL_MATRIX_DBG
nsPRINTF_DBG("::NsCallReadMatrices: Main process trying to load: %s\n", sFullFN);
#endif
try
{
    // try to load matrix
    Matrix mtmp(sFullFN);

    // check matrix & decide what to do
    if (NsProbMatrixIsGood(&mtmp) == TRUE)
    {
        // create matrix
        mdamatrices[nI].mpMatrix = new Matrix(&mtmp);
#ifndef NSCALL_MATRIX_DBG
        nsPRINTF_DBG("::NsCallReadMatrices: mtr # %d(%n, %n);\n",
                     mdamatrices[nI].mpMatrix->print());
#endif
    }
    eRetState = nsSTATE_START;
}
else
{
    nsPRINTF("Matrix \'%s\' don't obey conditions. Program stopped...\n", sFileNameIdx);
    eRetState = nsSTATE_EXIT;
    break;
}
catch(Matrix::exception e)
{
    nsPRINTF("::NsCallReadMatrices:Matrix EXPTION, code: %i\n", e.ec());
    eRetState = nsSTATE_EXIT;
    break;
}
}
else
    eRetState = nsSTATE_EXIT;
return eRetState;
}

void NscallUnReadMatricesAndDestroyStacks(NsMatrixData *mdamatrices, int nMatCount, NsStackData *sdpArray,
int nNumPrc)
{
    int nI;
    for (nI = 0; nI < nMatCount; nI++)
    {
        if (mdamatrices[nI].mpMatrix != NULL)
        {
            delete mdamatrices[nI].mpMatrix;
            mdamatrices[nI].mpMatrix = NULL;
        }
    }
    if (sdpArray != NULL)
    {
        for (nI = 0; nI < nNumPrc; nI++)
        {
            if (sdpArray[nI].sps != NULL)
            {
                delete sdpArray[nI].sps;
                sdpArray[nI].sps = NULL;
            }
        }
    }
}

// ****
// notes: Only process with ID == 0 must call me!
void NscallIMakeStepONE(Matrix &M, NsStack &sFirstF, NsCallLocalData *spData)
{
    int nCurrid;
    MPI_COMM_rank(MPI_COMM_WORLD, &nCurrid);
    // calculate distribution
    NsProbCalcDistribution(M, spData->rDelta, sFirstF);
    // out values
}

```

```

nsPRINTF((" Плотность распределения вер-ти времени выполнения (процесс %d):\n ", nCurID+1));
#ifndef nsIMM_PRINTF_ONLY
sFirstF.Print();
#endif
nsPRINTF((" Параметр m = %d (дельта = %.10f).\n\n", sFirstF.LastIdx(), spData->rDelta));
}

// -----
void NsCallMakestepTwo(Matrix &mQ2, NsStack &sFirst, NsStack &sSecond, int nMyID, NsCallLocalData *spData)
{
NsStack stmp(20);
// calculate distrib
NsProbCalcDistribution(mQ2, spData->rDelta, stmp);
nsPRINTF((" Плотность распределения вер-ти времени выполнения (процесс %d):\n ", nMyID+1));
stmp.Print();
nsPRINTF((" Параметр m = %d (дельта = %.10f).\n\n", stmp.LastIdx(), spData->rDelta));
// make convolution
NsProbCalcConvolution(sFirst, stmp, sSecond);
// out values
nsPRINTF((" Свертка (процесс %d):\n ", nMyID+1));
#ifndef nsIMM_PRINTF_ONLY
sSecond.Print();
#endif
nsPRINTF((" Параметр m = %d (дельта = %.10f).\n\n", sSecond.LastIdx(), spData->rDelta));
}

// -----
BOOL NsCallMain(int nNumPrc, int nMyID, char *cpFN)
{
int ni;
// distribution stacks
NsStack sFT1(20), sFT2(20), sFT_m(20), sFTN(20);
NsStack STimes(20);
int nMatrixCount = nNumPrc + 2;
NsCallLocalData *spData;
NsMatrixData *spMtrData;
NsStackData *spStkData;
if (nNumPrc < 3)
{
nsPRINTF_MM("::NsCallMain: Error: at least 3 processes must be used! exit..\n");
return TRUE;
}
// master process: prepare data
if (nMyID == 0)
{
spMtrData = new NsMatrixData[nMatrixCount];
spData = new NsCallLocalData[nMatrixCount];
// read matrix(es) from file(s)
spData[0].est = NsCallReadMatrices(spMtrData, nMatrixCount, cpFN);
if (spData[0].est != nsSTATE_START)
{
nsPRINTF_MM("::NsCallMain: User requested exit...\n");
for (ni = 1; ni < nNumPrc; ni++)
MPI_Send(spData, sizeof(NsCallLocalData), MPI_BYTE, ni, nsMSGTAG_DATA_A, MPI_COMM_WORLD);
delete [] spMtrData;
delete [] spData;
return TRUE;
}
// store data for all process
for (ni = 0; ni < nMatrixCount; ni++)
{
spData[ni].est = spData[0].est;
spData[ni].nMacols = spMtrData[ni].mpMatrix->cols();
spData[ni].nMakrows = spMtrData[ni].mpMatrix->rows();
spData[ni].rDelta = NSConfigGet()->rDelta;
}
// broadcast state & matrix
MO, (m1, m2, ..., mNumPrc), m{nNumPrc+1}
for (ni = 1; ni < nNumPrc; ni++)
{
MPI_Send(&(spData[ni+1]), sizeof(NsCallLocalData), MPI_BYTE, ni, nsMSGTAG_DATA_A, MPI_COMM_WORLD);
MPI_Send(spMtrData[ni].mpMatrix->data, spMtrData[ni].mpMatrix->cols() *
spMtrData[ni].mpMatrix->rows() *
sizeof(tMatrixElement), MPI_BYTE, ni, nsMSGTAG_MATRIX_ENTRY_A, MPI_COMM_WORLD);
}
// sent to slave1
MPI_Send(&(spData[1]), sizeof(NsCallLocalData), MPI_BYTE, 1, nsMSGTAG_DATA_B, MPI_COMM_WORLD);
MPI_Send(spMtrData[1].mpMatrix->data, spMtrData[1].mpMatrix->cols() * spMtrData[1].mpMatrix->rows() *
sizeof(tMatrixElement), MPI_BYTE, 1, nsMSGTAG_MATRIX_ENTRY_B, MPI_COMM_WORLD);
}

```

```

// send to slave2
MPI_Send(&(spData[nNumPrc+1]), sizeof(NsCallLocalData), MPI_BYTE, 2, nmsgtag_data_B, MPI_COMM_WORLD);
MPI_Send(spMtrData[nNumPrc+1].mpMatrix->data, spMtrData[nNumPrc+1].mpMatrix->cols() * sizeof(tMatrixElement), MPI_BYTE, 2, nmsgtag_matrix_entry_B, MPI_COMM_WORLD);
}
else if (nMyID == 1 || nMyID == 2)
{
    spMtrData = new NsMatrixData[2];
    spData = new NsCallLocalData[2];
    MPI_Status sstat;

    // broadcast state
    MPI_Recv(&(spData[0]), sizeof(NsCallLocalData), MPI_BYTE, 0, nmsgtag_data_A, MPI_COMM_WORLD, &sstat);
    if (spData->est != nsSTATE_START)
    {
        delete [] spMtrData;
        delete [] spData;
        return TRUE;
    }

    // Create matrix
    spMtrData[0].mpMatrix = new Matrix(spData[0].nMarows, spData[0].nmcols);
    MPI_Recv(&(spMtrData[0].mpMatrix->data, spMtrData[0].mpMatrix->cols() * spMtrData[0].mpMatrix->rows() * sizeof(tMatrixElement)), MPI_BYTE, 0, nmsgtag_matrix_entry_A, MPI_COMM_WORLD, &sstat);

    // 2
    MPI_Recv(&(spData[1]), sizeof(NsCallLocalData), MPI_BYTE, 0, nmsgtag_data_B, MPI_COMM_WORLD, &sstat);
    spMtrData[1].mpMatrix = new Matrix(spData[1].nMarows, spData[1].nmcols);
    MPI_Recv(&(spMtrData[1].mpMatrix->data, spMtrData[1].mpMatrix->cols() * spMtrData[1].mpMatrix->rows() * sizeof(tMatrixElement)), MPI_BYTE, 0, nmsgtag_matrix_entry_B, MPI_COMM_WORLD, &sstat);
}
else
{
    spMtrData = new NsMatrixData[1];
    spData = new NsCallLocalData[1];
    MPI_Status sstat;

    // broadcast state
    MPI_Recv(&(spData[0]), sizeof(NsCallLocalData), MPI_BYTE, 0, nmsgtag_data_A, MPI_COMM_WORLD, &sstat);
    if (spData->est != nsSTATE_START)
    {
        delete [] spMtrData;
        delete [] spData;
        return TRUE;
    }

    // Create matrix
    spMtrData[0].mpMatrix = new Matrix(spData[0].nMarows, spData[0].nmcols);
    MPI_Recv(&(spMtrData[0].mpMatrix->data, spMtrData[0].mpMatrix->cols() * spMtrData[0].mpMatrix->rows() * sizeof(tMatrixElement)), MPI_BYTE, 0, nmsgtag_matrix_entry_A, MPI_COMM_WORLD, &sstat);
}

// ONE #####
if (nMyID == 0)
{
    spstkData = new NsStackData[nNumPrc];
    nsprintf(("##### 3tran %1 #####\n"));
}

sTimes += MPI_Wtime();

### first step here
NsCallMakeStepOne((spMtrData[0].mpMatrix), sft1, &spData[0]);

sTimes += MPI_Wtime();

if (nMyID == 0)
{
    int ntmpsize;
    MPI_Status sstat;

    for (nI = 1; nI < nNumPrc; nI++)
    {
        MPI_Recv(&ntmpsize, 1, MPI_INT, MPI_ANY_SOURCE, nmsgtag_slave_distsize_A, MPI_COMM_WORLD, &sstat);
        spstkData[nI].sps = new NsStack(ntmpsize);

        MPI_Recv(&spstkData[nI].sps->pdata, sizeof(NsReal) * ntmpsize, MPI_BYTE, MPI_ANY_SOURCE,
        nmsgtag_slave_distentry_A, MPI_COMM_WORLD, &sstat);
        spstkData[nI].sps->SetLastIdx(ntmpSize - 1);
    }
}
else
{
    int nsize = sft1.LastIdx() + 1;
    MPI_Send(&(nsize), 1, MPI_INT, 0, nmsgtag_slave_distsize_A, MPI_COMM_WORLD);
    MPI_Send(&sft1.pData, sizeof(NsReal) * nsize, MPI_BYTE, 0, nmsgtag_slave_distentry_A, MPI_COMM_WORLD);
}

sTimes += MPI_Wtime();

if (nMyID == 0)
{
    int ntmpsize;

```

```

MPI_Status sStat;
NsStack sStackTmp(20);
nsPRINTF("##### Этап №2 #####\n");
for (nI = 1; nI < nNumPrc; nI++)
{
    sStackTmp = *(spStkData[nI].sps);
    NsProbCalcConvolution(sFT1, sStackTmp, *(spStkData[nI].sps));
}
sTimes += MPI_Wtime();

MPI_Recv(&nTmpSize, 1, MPI_INT, 1, nsMSGTAG_SLAVE_DISTSIZE_B, MPI_COMM_WORLD, &sStat);
spStkData[0].sps = new NSStack(nTmpSize);
MPI_Recv(spStkData[0].sps->pData, sizeof(NsReal) * nTmpSize, MPI_BYTE, 1, nsMSGTAG_SLAVE_DISTENTRY_B,
MPI_COMM_WORLD, &sStat);
spStkData[0].sps->SetLastIdx(nTmpSize - 1);

// calculate last conv
sStackTmp = *(spStkData[0].sps);
NsProbCalcConvolution(sFT1, sStackTmp, *(spStkData[0].sps));

MPI_Recv(&nTmpSize, 1, MPI_INT, 2, nsMSGTAG_SLAVE_DISTSIZE_B, MPI_COMM_WORLD, &sStat);
sFT2.SetLastIdx(nTmpSize - 1);
MPI_Recv(spStkData[0].sps->pData, sizeof(NsReal) * nTmpSize, MPI_BYTE, 2, nsMSGTAG_SLAVE_DISTENTRY_B,
MPI_COMM_WORLD, &sStat);
}
else if (nMyID == 1 || nMyID == 2)
{
    nsCallMakeStepONE(*spMtrData[1].mpMatrix), sFT2, &spData[1]);
    sTimes += MPI_Wtime();
    {
        int nSize = sFT2.LastIdx() + 1;
        MPI_Send(&nSize, 1, MPI_INT, 0, nsMSGTAG_SLAVE_DISTSIZE_B, MPI_COMM_WORLD);
        MPI_Send(sFT2.pData, sizeof(NsReal) * nSize, MPI_BYTE, 0, nsMSGTAG_SLAVE_DISTENTRY_B,
MPI_COMM_WORLD);
    }
}
else
{
    sTimes += MPI_Wtime();
}

sTimes += MPI_Wtime();

// THREE #####
if (nMyID == 0)
{
    NsReal rAvg;
    NsReal rDisp;

    NsProbCalcANDUnion(spStkData, nNumPrc, SFT_M);
    //
    nsPRINTF("##### Этап №3 #####\n AND результат:\n ");
#ifndef nsIMM_PRINTF_ONLY
    SFT_M.Println();
#endif

#If defined(nsCALL_MATRIX_DBG) && defined(nsDEBUG_MSG)
    nsPRINTF_DBG("!! Step 3, MyID = %d, Matrix: \n", nMyID);
    spMtrData[nNumPrc+1].mpMatrix->print();
#endif

    NsProbCalcConvolution(sFT2, SFT_M, SFT_N);

    // calculate M & D
    rAvg = SFT_N.getAverage();
    rDisp = SFT_N.getDispersion(rAvg);

    sTimes += MPI_Wtime();
    nsPRINTF_INN(" Среднее = %.4lf, дисперсия = %.4lf\n##### Вычисления завершены ##### (cols = %d)\n",
    rAvg, rDisp, spMtrData[0].mpMatrix->cols());
}
else
{
    sTimes += MPI_Wtime();
}

// timing statistic
MPI_Barrier(MPI_COMM_WORLD);
nsPRINTF_INN("Процесс %d, дельта: %.8lf, тайминг: \n", nMyID, sTimes[sTimes.LastIdx()]-sTimes[0]));
sTimes.Print();
nsPRINTF_INN("коммуникации суммарно: %.8lf, вычисления суммарно: %.8lf\n\n",
sTimes[2]-sTimes[1] + sTimes[4]-sTimes[3], sTimes[1]-sTimes[0] + sTimes[3]-sTimes[2] + sTimes[5]-sTimes[4]));

// destroy objects
if (nMyID == 0)
{
    NSCallUnReadMatricesAndDestroyStacks(spMtrData, nMatrixCount, spStkData, nNumPrc);
}
else if (nMyID == 1 || nMyID == 2)
{
    NScallUnReadMatricesAndDestroyStacks(spMtrData, 2, NULL, 0);
}

```

```

else
{
    NSCallUnReadMatricesAndDestroyStacks(spMtrData, 1, NULL, 0);
}
delete [] spMtrData;
delete [] spData;

#ifndef nsCALL_NO_INPUT
return FALSE;
#else
return TRUE;
#endif
}

```

## 2. Обработчик выходных данных прототипов на языке Perl

```

#!/usr/local/bin/perl

# nsskel output parser
# (c) Nnick V2.1

if (open(timelog, $ARGV[0]))
{
    @timelog = <timelog>;
    close timelog;
}

$ci = -1;
$iter = 0;
$curr_cols = 0;
$tti = 0;

foreach (@timelog)
{
    ##### вычисления завершены ##### (cols = 5)
    if (/^#####.*cols = ([0-9]+)\)/)
    {
        $new_cols = $1;

        # print "col \"$iter.\" \"$ci\".\n";
        for ($ni = 0; $ni < $tti; $ni++)
        {
            # 10 == max prc num
            for ($np = 0; $np < 10; $np++)
            {
                if ($mentr->[$ci][$iter]->{PRC}->[$np]->{TIME})
                {
                    if (abs ($mentr->[$ci][$iter]->{PRC}->[$np]->{TIME} - $timetmp->[$ni]->{COMM} - $timetmp->[$ni]->{CALC}) < 0.0000001)
                    {
                        $mentr->[$ci][$iter]->{PRC}->[$np]->{TIME_COMM} = $timetmp->[$ni]->{COMM};
                        $mentr->[$ci][$iter]->{PRC}->[$np]->{TIME_CALC} = $timetmp->[$ni]->{CALC};
                    }
                }
            }
            $tti = 0;
        }

        if ($new_cols ne $curr_cols)
        {
            $curr_cols = $new_cols;
            $iter = 0;
            $ci++;
        }
        else
        {
            $iter++;
        }

        $mentr->[$ci][$iter]->{COLS} = $curr_cols;
    }
    #процесс 0, дельта: 0.16754979, тайминг:
    elsif (/^Процесс ([0-9]+), дельта: ([0-9]+\.[0-9]+)\)/
    {
        $mentr->[$ci][$iter]->{PRC}->[$1]->{TIME} = $2;
    }
    # print "Prc = $1, time: ", $mentr->[$ci][$iter]->{PRC}->[$1]->{TIME} ."\n";
    #коммуникации суммарно: 0.16546237, вычисления суммарно: 0.00208742
    elsif (/^Коммуникации суммарно: ([0-9]+\.[0-9]+), вычисления суммарно: ([0-9]+\.[0-9]+)\)/
    {
        $timetmp->[$tti]->{COMM} = $1;
        $timetmp->[$tti]->{CALC} = $2;
        $tti++;
    }
}

#if (!$mentr->[$ci])
$ci = 0;

```

```

# По всем размерам матриц
while($MEntr->[$c1])
{
    # по всем итерациям
    $iter = 1;
    while($MEntr->[$c1][$iter])
    {
        # По всем процессам
        $nP = 0;
        if ($MEntr->[$c1][$iter]->{PRC}->[0]->{TIME})
        {
            while($MEntr->[$c1][$iter]->{PRC}->[$nP]->{TIME})
            {
                print "ASSERT:: NO time at processor \n" if !($MEntr->[$c1][$iter]->{PRC}->[$nP]->{TIME_COMM});
                print "ASSERT:: NO time at processor \n" if !($MEntr->[$c1][$iter]->{PRC}->[$nP]->{TIME_CALC});
                $MEntr->[$c1][0]->{PRC}->[$nP]->{TIME} += $MEntr->[$c1][$iter]->{PRC}->[$nP]->{TIME};
                $MEntr->[$c1][0]->{PRC}->[$nP]->{TIME_COMM} += $MEntr->[$c1][$iter]->{PRC}->[$nP]->{TIME_COMM};
                $MEntr->[$c1][0]->{PRC}->[$nP]->{TIME_CALC} += $MEntr->[$c1][$iter]->{PRC}->[$nP]->{TIME_CALC};

                $nP++;
            }
            print "Number of prc ". $nP ."\n";
        }
        $iter++;
    }
    # По всем процессам
    $nP = 0;
    if ($MEntr->[$c1][0]->{PRC}->[0]->{TIME})
    {
        printf(" m: %3d " . $MEntr->[$c1][0]->{COLS});
        while($MEntr->[$c1][0]->{PRC}->[$nP]->{TIME})
        {
            $MEntr->[$c1][0]->{PRC}->[$nP]->{TIME} /= $iter;
            $MEntr->[$c1][0]->{PRC}->[$nP]->{TIME_COMM} /= $iter;
            $MEntr->[$c1][0]->{PRC}->[$nP]->{TIME_CALC} /= $iter;

            printf("p: %12.9f %12.9f %12.9f " . $nP, $MEntr->[$c1][0]->{PRC}->[$nP]->{TIME}, $MEntr->[$c1][0]->{PRC}->[$nP]->{TIME_COMM}, $MEntr->[$c1][0]->{PRC}->[$nP]->{TIME_CALC});
            $nP++;
        }
        print "\n";
    }
    print "ASSERT:: Number of iterations: ".$iter."\n" if $iter ne 4;
    $c1++;
}

```

### 3. Текст программы MathCad для прототипа объединения подпрограмм по логической функции «И»

Стохастическая матрица вероятностей переходов:

$$P := \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Точность:

$$\delta := 0.00000000001$$

Число процессов:

$$N := 2$$

$$\text{findK}(M, \delta) := \left| \begin{array}{l} i \leftarrow 1 \\ \text{while } \left[ 1 - (P^{i-1})_{0,\text{cols}(P)-1} > \delta \right] \\ \quad i \leftarrow i + 1 \end{array} \right|$$

$$K := \text{findK}(P, \delta)$$

Число элементов распределения (необходимых для достижения заданной точности):

$$K = 3$$

Плотность распределения вероятностей времени выполнения:

$$i := 1..K$$

$$fT1_i := (P^i)_{0,\text{cols}(P)-1} - (P^{i-1})_{0,\text{cols}(P)-1}$$

$$fT1_0 := 0$$

$$i := K + 1..2\cdot K$$

$$fT1_i := 0$$

Результат вычисления:

$$fT_1^T = (0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0)$$

Свертка для параллельных процессов:

$j := 0..2 \cdot K$

$$fT_2 := \sum_{i=0}^j fT_1 \cdot fT_{j-i}$$

Результат вычисления:

$$fT_2^T = (0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0)$$

Вычисление объединения результатов по AND:

$j := 1..2 \cdot K$

$$fT_N := \prod_{i=0}^N \sum_{m=0}^j fT_m - \prod_{i=0}^N \sum_{m=0}^{j-1} fT_m$$

Результат вычисления объединения AND:

$$fT_N^T = (0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0)$$

Итоговое распределение - финальная свертка:

$$fN := \sum_{i=0}^j fT_1 \cdot fT_{j-i}$$

$$fN^T = (0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1)$$

Матожидание:

$2 \cdot K$

$$M := \sum_{j=0}^{2 \cdot K} j \cdot fN_j$$

$M = 6$

Дисперсия:

$2 \cdot K$

$$\sum_{j=0}^{2 \cdot K} (j - M)^2 \cdot fN_j = 0$$

4. Текст программы MathCad для прототипа объединения подпрограмм по логической функции «ИЛИ»

Стochastic матрица вероятностей переходов:

$$P := \begin{pmatrix} 0 & 0.5 & 0 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0.5 & 0.5 \\ 0.5 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Точность:

$$\delta := 0.0000001$$

Число процессов:

$$N := 3$$

$$\text{findK}(M, \delta) := \left| \begin{array}{l} i \leftarrow 1 \\ \text{while } \left[ 1 - \left( P^{i-1} \right)_{0, \text{cols}(P)-1} > \delta \right] \\ \quad i \leftarrow i + 1 \end{array} \right|$$

$$K := \text{findK}(P, \delta)$$

Число элементов распределения (необходимых для достижения заданной точности):

$$K = 25$$

$j \geq 1..2-K$

$$fT_j := (P^j)_{0, \text{cols}(P)-1} - (P^{j-1})_{0, \text{cols}(P)-1}$$

$$fT_0 = 0$$

	0	1	2	3	4	5	6
0	0	0.5	0.25	0.125	0.063	0.031	0.016

Вычисление объединения результатов по OR:

$$\Omega_j := \left[ 1 - (P^{j-1})_{0, \text{cols}(P)-1} \right]^N - \left[ 1 - (P^j)_{0, \text{cols}(P)-1} \right]^N$$

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0.875	0.108	0.014	0.002	0	0	0	0	0	0	0

Матожидание:

$$M := \sum_{j=0}^{2-K} j \cdot \Omega_j$$

$$M = 1.143$$

Дисперсия:

$$\sum_{j=0}^{2-K} (j - M)^2 \cdot \Omega_j = 0.163$$

# Программная документация на программное обеспечение для определения статистических характеристик обнаружения и отражения угроз систем защиты информации третьего класса защищенности

Программное обеспечение разработано в интегрированной среде Microsoft Visual Studio 6.0 с использованием библиотеки MPI, поставляемой пакетом MPICH.NT 1.2.5, и реализовано на персональном компьютере под управлением операционной системы Microsoft Windows XP. Визуальный интерфейс реализован в системе Borland Delphi 7.0. В соответствии с ГОСТ 19.101-77 представлены следующие описания:

- 1) Спецификация (ГОСТ 19.202-78).
  - 2) Техническое задание (ГОСТ 19.201-78).
  - 3) Текст программы (ГОСТ 19.401-78).
  - 4) Описание программы (ГОСТ 19.402-78).
  - 5) Программа и методика испытания (ГОСТ 19.301-78).

## Спецификация программы

P.II.68145-06

## **Техническое задание**

P.P.68145-06

### **Введение**

Данная программа предназначена для оценки динамических характеристик параллельных вычислений.

### **Основания для разработки**

1) документ:

настоящее учебное пособие;

2) выдано:

ГОУ ВПО СПбГПУ, факультетом технической кибернетики, профессором Птицыной Л.К., 1 сентября 2005г.;

3) наименование:

«Разработка программы для оценки статистических характеристик обнаружения и отражения угроз для автоматизированных систем третьего класса защищенности в базисе функций MPI».

### **Назначение разработки**

Программа предназначена для определения статистических характеристик обнаружения и отражения угроз автоматизированной системой третьего класса защищенности.

### **Требования к программе или программному изделию**

#### ***Требования к функциональным характеристикам.***

Программа должна реализовать следующие функции:

- Вычисление плотностей распределения вероятностей дискретного времени выполнения отдельных подпроцессов функционирования автоматизированной системы третьего класса защищенности;
- Вычисление плотностей распределения вероятностей дискретного времени окончания выполнения всех подпроцессов функционирования автоматизированной системы третьего класса защищенности;
- Вычисление базовых статистических характеристик через плотности распределения вероятностей дискретного времени окончания выполнения решающих подпроцессов функционирования автоматизированной системы третьего класса защищенности;

- Обеспечение графического интерфейса пользователя для предоставления возможностей выбора исходных параметров модели.

#### *Требования к составу и параметрам технических средств.*

Разрабатываемая программа должна функционировать на компьютерах, которые по своим техническим параметрам удовлетворяют требованиям ОС семейства Windows, с установленным пакетом MPICH.NT 1.2.5.

#### *Требования к информационной и программной совместимости.*

Программа должна быть написана на языке С в базисе функций библиотеки MPI, трансляция программы осуществляется с помощью компилятора Visual Studio 6.0. Программа должна работать под управлением ОС Windows XP с установленным пакетом MPICH.NT 1.2.5.

#### **Требования к программной документации**

Программная документация должна соответствовать ГОСТ 19.201-78.

#### **Порядок контроля и приемки**

По окончании написания программы исполнитель проводит тестирование программы в лабораторных условиях и устраняет все обнаруженные недочеты.

## Описание программы

Р.П.6815-06.13.01

### Общие сведения

Программа должна выполнять вычисление плотности распределения вероятностей времени принятия решения относительно наступления каждого из пяти завершающих событий в базовой модели процессов обнаружения и отражения угроз для автоматизированных систем 3-го класса защищенности, представленной на рис. 5.2. Программа должна выполнять параллельное вычисление промежуточных данных с использованием функций библиотеки MPI 1.1. Входными данными для программы являются:

- Времена выполнения каждого подпроцесса графа:

$$\begin{aligned} k_1 &= 1, \dots, k_{\text{обн.отр.ид}}; k_2 = 1, \dots, k_{\text{необн.отр.ид}}; k_3 = 1, \dots, k_{\text{необн.неотр.ид}}; \\ k_4 &= 1, \dots, k_{\text{обн.отр.цел}}; k_5 = 1, \dots, k_{\text{обн.неотр.цел}}; k_6 = 1, \dots, k_{\text{необн.неотр.цел}}; \\ k_7 &= 1, \dots, k_{\text{обн.рег.уч}}; k_8 = 1, \dots, k_{\text{необн.рег.уч}}; k_9 = 1, \dots, k_{\text{обн.рег.уч}}; \\ k_{10} &= 1, \dots, k_{\text{необн.рег.уч}}; k_{11} = 1, \dots, k_{\text{акт.реакц.1}}; k_{12} = 1, \dots, k_{\text{насс.реакц.1}}; \\ k_{13} &= 1, \dots, k_{\text{акт.реакц.2}}; k_{14} = 1, \dots, k_{\text{насс.реакц.2}}; k_{15} = 1, \dots, k_{\text{акт.реакц.3}}; \\ k_{16} &= 1, \dots, k_{\text{насс.реакц.3}}; k_{17} = 1, \dots, k_{\text{акт.реакц.4}}; k_{18} = 1, \dots, k_{\text{насс.реакц.4}}; \\ k_{19} &= 1, \dots, k_{\text{зад.реш.0}}; \dots; k_{28} = 1, \dots, k_{\text{зад.реш.9}}. \end{aligned}$$

- Вероятности переходов между подпроцессами:  $p_{ld1}, p_{ld2}, p_{reg1}, p_{reg2}, p_{react1}, p_{react2}, p_{react3}, p_{react4}, p_{init1}, p_{init2}$ .

- Законы распределения для подпроцессов графа:  
экспоненциальный, равномерный, детерминированный.

Выходные данные представляют собой плотности распределения вероятностей для каждого итогового решения:

$$\begin{aligned} f_{\text{акт.обн.отр.}}(k_1), k_1 &= 1, \dots, k_{\text{акт.обн.отр.}}; \\ f_{\text{насс.обн.отр.}}(k_2), k_2 &= 1, \dots, k_{\text{насс.обн.отр.}}; \\ f_{\text{акт.обн.неотр.}}(k_3), k_3 &= 1, \dots, k_{\text{акт.обн.неотр.}}; \\ f_{\text{насс.обн.неотр.}}(k_4), k_4 &= 1, \dots, k_{\text{насс.обн.неотр.}}; \\ f_{\text{необн.неотр.}}(k_5), k_5 &= 1, \dots, k_{\text{необн.неотр.}} \end{aligned}$$

### Функциональное назначение

Разработанное программное обеспечение реализует следующие функции:

- Вычисление плотностей распределения вероятностей дискретного времени выполнения отдельных подпроцессов функционирования автоматизированной системы третьего класса защищенности;

- Вычисление плотностей распределения вероятностей дискретного времени окончания выполнения всех подпроцессов функционирования автоматизированной системы третьего класса защищенности;
- Вычисление базовых статистических характеристик через плотности распределения вероятностей дискретного времени окончания выполнения решающих подпроцессов функционирования автоматизированной системы третьего класса защищенности;
- Обеспечение графического интерфейса пользователя для предоставления возможностей выбора исходных параметров модели.

Для вычисления характеристик применяются аналитические соотношения, выведенные с помощью метода свертки для последовательных подпроцессов в составе параллельных, для параллельных подпроцессов, объединяемых по логической функции «И», и для параллельных подпроцессов, объединяемых по логической функции «ИЛИ». Указанный набор функций необходим и достаточен для моделирования процесса функционирования автоматизированной системы третьего класса защищенности.

При оценке искомых характеристик задействованы следующие функции библиотеки MPI: `MPI_Bcast()`, `MPI_Reduce()`, `MPI_Send()`, `MPI_Recv()`.

Обозначения подпроцессов приводятся как  $f(k)$ , так как соответствующие характеристики считаются по одному правилу, но с разными временами. Параметры типовых функций MPI приведены в Приложении 1.

### **Описание логической структуры**

В данном случае предлагается пятиэтапный вариант логической структуры программы, приведенный на рис. ПЗ.1.

На этапе №1 происходит инициализация всех данных, необходимых для вычислений: массива  $K_{max\ i}$ ,  $i = 1,..,28$ ; вероятностей  $p_{id1}$ ,  $p_{id2}$ ,  $p_{reg1}$ ,  $p_{reg2}$ ,  $p_{react1}$ ,  $p_{react2}$ ,  $p_{react3}$ ,  $p_{react4}$ ,  $p_{ini1}$ ,  $p_{ini2}$ . Последовательно вычисляется и заполняется массив  $\tilde{k}[11]$ . Инициализируется MPI. Главный процесс посыпает остальным границы выполнения цикла и данные для вычисления  $f_i(k_i)$ . Рассылка осуществляется функцией `MPI_Bcast`.

На этапе №2 происходит параллельное формирование массива значений для каждого  $i$   $f_i(k_i)$ .

На этапе №3 происходит сбор результатов с помощью функции MPI\_Reduce. Рассылаются данные для вычисления  $f_i(\tilde{k}_i)$ . Рассылка осуществляется функцией MPI\_Bcast.

На этапе №4 реализуется параллельное вычисление  $f_i(\tilde{k}_i)$ .

На этапе №5 происходит сбор результатов с использованием функции MPI\_Reduce вычисление интервалов  $\hat{k}_i$ . Рассылаются данные для вычисления  $q_i(\hat{k}_i)$ . Рассылка осуществляется функцией MPI\_Bcast.

На этапе №6 реализуется параллельное вычисление  $q_i(\hat{k}_i)$ .

На этапе №7 происходит сбор результатов с использованием функции MPI\_Reduce. Вычисляются окончательные результаты:

$f_{\text{акт.обн.отр.}}(k_1), k_1=1, \dots, k_{\text{акт.обн.отр.}}$ ;

$f_{\text{пасс.обн.отр.}}(k_2), k_2=1, \dots, k_{\text{пасс.обн.отр.}}$ ;

$f_{\text{акт.обн.неотр.}}(k_3), k_3=1, \dots, k_{\text{акт.обн.неотр.}}$ ;

$f_{\text{пасс.обн.неотр.}}(k_4), k_4=1, \dots, k_{\text{пасс.обн.неотр.}}$ ;

$f_{\text{необн.неотр.}}(k_5), k_5=1, \dots, k_{\text{необн.неотр.}}$ .

## Используемые технические средства

Для работы программы необходимы:

1. Оборудование:

- компьютер с операционной системой семейства Windows;
- локальная компьютерная сеть.

2. Операционная система:

- Microsoft Windows XP
- Microsoft Windows 2000
- Microsoft Windows NT .

3. Установленное программное обеспечение:

- на каждой машине в сети должен быть установлен пакет MPICH.NT 1.2.5.

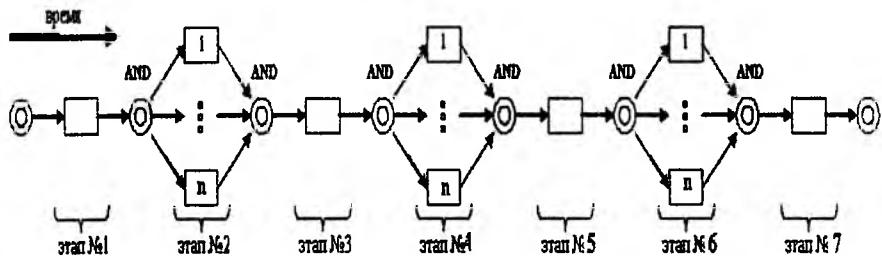


Рис. П3.1. Логическая структура программы

### Вызов и загрузка

Программа может запускаться любым способом при загруженной ОС Windows. В качестве примера можно привести следующие варианты:

- из окна программы Проводник;
- из программ Far, Norton Commander, Volcov Commander.

### Входные данные

Входными данными для программы являются времена выполнения каждого подпроцесса графа:

$$\begin{aligned}
 k_1 &= 1, \dots, k_{\text{обн.отр.ид}}; \quad k_2 = 1, \dots, k_{\text{необн.отр.ид}}; \quad k_3 = 1, \dots, k_{\text{необн.неотр.ид}}; \quad k_4 = 1, \dots, k_{\text{обн.отр.чел}}; \\
 k_5 &= 1, \dots, k_{\text{обн.неотр.чел}}; \quad k_6 = 1, \dots, k_{\text{необн.неотр.чел}}; \\
 k_7 &= 1, \dots, k_{\text{обн.рег.уч}}; \quad k_8 = 1, \dots, k_{\text{необн.рег.уч}}; \quad k_9 = 1, \dots, k_{\text{обн.рег.уч}}; \\
 k_{10} &= 1, \dots, k_{\text{необн.рег.уч}}; \quad k_{11} = 1, \dots, k_{\text{акт.реакц.1}}; \quad k_{12} = 1, \dots, k_{\text{насс.реакц.1}}; \\
 k_{13} &= 1, \dots, k_{\text{акт.реакц.2}}; \quad k_{14} = 1, \dots, k_{\text{насс.реакц.2}}; \quad k_{15} = 1, \dots, k_{\text{акт.реакц.3}}; \\
 k_{16} &= 1, \dots, k_{\text{насс.реакц.3}}; \quad k_{17} = 1, \dots, k_{\text{акт.реакц.4}}; \quad k_{18} = 1, \dots, k_{\text{насс.реакц.4}}; \\
 k_{19} &= 1, \dots, k_{\text{зад.реш.0}}; \dots; \quad k_{28} = 1, \dots, k_{\text{зад.реш.9}}.
 \end{aligned}$$

Вероятности переходов между подпроцессами:  $p_{id1}, p_{id2}, p_{reg1}, p_{reg2}, p_{react1}, p_{react2}, p_{react3}, p_{react4}, p_{int1}, p_{int2}$ .

Указанные данные должны храниться в файлах "Kmax.txt" и "Prob.txt". Файл "Kmax.txt" содержит числа типа int разделенные пробелами. Данные располагаются в порядке возрастания:  $k1, \dots, k28$ . Файл "Prob.txt" содержит данные типа int. Числа разделены пробелами и располагаются в следующем порядке:

$$P_{id1} \ P_{id2} \ P_{reg1} \ P_{reg2} \ P_{int1} \ P_{int2} \ P_{react1} \ P_{react2} \ P_{react3} \ P_{react4}.$$

Входным параметром является и закон распределения плотности вероятности в подпроцессах:

0 – экспоненциальный; 1 – равномерный; 2 – детерминированный.

### Выходные данные

На выходе программа выводит значения плотностей распределения вероятностей для пяти итоговых решений  $f_{акт.обн.отр.}(k_1)$ ,  $f_{насс.обн.отр.}(k_2)$ ,  $f_{акт.обн.неотр.}(k_3)$ ,  $f_{насс.обн.неотр.}(k_4)$ ,  $f_{необн.неотр.}(k_5)$ .

При запуске программы без графического интерфейса полученные данные выводятся на экран. При работе с графическим интерфейсом в директории с программой формируются файлы 1.res, 2.res, 3.res, 4.res, 5.res, содержащие выходные данные.

Файл 1.res соответствует выходу графа  $f_{акт.обн.отр.}(k_1)$ .

Файл 2.res соответствует выходу графа  $f_{насс.обн.отр.}(k_2)$ .

Файл 3.res соответствует выходу графа  $f_{акт.обн.неотр.}(k_3)$ .

Файл 4.res соответствует выходу графа  $f_{насс.обн.неотр.}(k_4)$ .

Файл 5.res соответствует выходу графа  $f_{необн.неотр.}(k_5)$ .

В файлах содержится следующая информация:

Число\_получаемых\_значений Минимальное\_время выполнения выходной функции Максимальное\_время\_ выполнения\_ выходной функции  
Числовые\_значения

Пример содержимого выходного файла приведен ниже.

31	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
0.000971	0.000822	0.000695	0.000589	0.000498	0.000422	0.000357	0.000302	0.000256	0.000217	0.000183	0.000155	0.000131	0.000111	0.000094	0.000080	0.000067	0.000057	0.000048	0.000041	0.000035	0.000029	0.000025	0.000021	0.000018	0.000015	0.000013	0.000011	0.000009	0.000008	0.000007	

Графический интерфейс позволяет просмотреть полученные данные с помощью графиков (рис.ПЗ.1).

# **Программа и методика испытаний программы**

**Р.П.6815-06.51.01**

## **Объект испытаний**

Объектом испытаний является построенная базовая модель процессов обнаружения и отражения угроз для автоматизированных систем третьего класса защищенности, которая расширяет пространство моделирования систем защиты информации.

## **Цель испытаний**

Целью проведения испытаний является проверка соответствия функциональных характеристик программы требованиям, указанным в техническом задании. Согласование результатов выполнения созданного программного средства с данными, полученными с применением пакета MathCad. Выявление зависимости времени работы программы от размерности входных данных

## **Требования к программе**

Программа должна реализовать следующие функции:

- Вычисление плотностей распределения вероятностей дискретного времени выполнения отдельных подпроцессов функционирования автоматизированной системы третьего класса защищенности;
- Вычисление плотностей распределения вероятностей дискретного времени окончания выполнения всех подпроцессов функционирования автоматизированной системы третьего класса защищенности;
- Вычисление базовых статистических характеристик через плотности распределения вероятностей дискретного времени окончания выполнения решающих подпроцессов функционирования автоматизированной системы третьего класса защищенности;
- Обеспечение графического интерфейса пользователя для предоставления возможностей выбора исходных параметров модели.

## **Требования к программной документации**

Программная документация должна соответствовать ГОСТ 19.301-

## **Состав и порядок испытаний**

Программа должна выполняться на компьютере с архитектурой x86 со следующими характеристиками:

**1) Оборудование:**

- компьютер с операционной системой семейства Windows;
- локальная компьютерная сеть.

**2) Операционная система:**

- Microsoft Windows XP;
- Microsoft Windows 2000;
- Microsoft Windows NT.

**3) Установленное программное обеспечение:**

- на каждой машине в сети должен быть установлен пакет MPICH.NT 1.2.5.

## **Методы испытаний**

Подтверждение корректности функционирования разработанной программы осуществляется посредством сопоставления получаемых значений динамических характеристик со значениями, найденными с использованием пакета MathCad на тестовых наборах данных, соответствующих альтернативным вариантам. Тестирование проводится на локальной машине и в сети, используя графический интерфейс.

Для тестирования проводятся следующие эксперименты:

1. Функции плотности распределения времени выполнения каждого подпроцесса графа представлены экспоненциальными распределениями. Диапазон времени выполнения каждого подпроцесса  $[1, \dots, K_{max_i}]$ ;
2. Функции плотности распределения времени выполнения каждого подпроцесса графа представлены экспоненциальным распределением. Диапазон времени выполнения каждого подпроцесса  $[1, 2]$ ;
3. Функции плотности распределения времени выполнения каждого подпроцесса графа представлены как  $f=1/K_{max_i}$ . Диапазон времени выполнения каждого блока  $[1, \dots, K_{max_i}]$ ;
4. Функции плотности распределения времени выполнения каждого подпроцесса графа представлены как  $f(K_i)=1$  для  $K_i=random(1..K_{max_i})$  и  $f(K_i)=0$  для  $K_i \neq K_i$ . В этом случае при больших разбросах временных

интервалов нулевой результат на выходе будет достаточно частым, поэтому время выполнения подпроцесса лежит в интервале [1,2].

Для тестирования использовались входные данные, показанные в табл.П3.1.

Таблица П3.1.

**Исходные данные для проведения эксперимента**

Максимальное значение хранилища экспериментальных подпроцессов	№ эксперимента	№ эксперимента	Вероятности	Значения вероятностей
	1,3	2,4		
k1	1..4	1..2	P <sub>01</sub>	0.2
k2	1..4	1..2	P <sub>02</sub>	0.4
k3	1..4	1..2	P <sub>03</sub>	0.4
k4	1..4	1..2	P <sub>04</sub>	0.1
k5	1..4	1..2	P <sub>05</sub>	0.2
k6	1..4	1..2	P <sub>06</sub>	0.5
k7	1..4	1..2	P <sub>07</sub>	0.5
k8	1..4	1..2	P <sub>08</sub>	0.2
k9	1..4	1..2	P <sub>09</sub>	0.7
k10	1..4	1..2	P <sub>10</sub>	0.3
В х о д к и ж е к а и и с	k11	1..4	1..2	
	k12	1..4	1..2	
	k13	1..4	1..3	
	k14	1..4	1..2	
	k15	1..4	1..2	
	k16	1..4	1..2	
	k17	1..4	1..2	
	k18	1..4	1..2	
	k19	1..4	1..2	
	k20	1..4	1..2	
Х и з и с т и и	k21	1..4	1..2	
	k22	1..4	1..2	
	k23	1..4	1..2	
	k24	1..4	1..2	
	k25	1..4	1..2	
	k26	1..4	1..2	
	k27	1..4	1..2	
	k28	1..4	1..2	

При испытаниях следует исследовать временные затраты на выполнение программы локально и в сети.

### Результаты экспериментов

Данные, полученные в результате экспериментов с использованием библиотеки MPI и с использованием пакета MathCad, получились идентичными. В табл. П3.2, П3.3, П3.4, П3.5 приведены их значения.

Таблица П3.2.

## Результаты работы программы для эксперимента 1

Результат работы программы	Плотность распределения вероятности времени выполнения программы защиты информации	Математическое ожидание	Дисперсия	Асимметрия	Эксцесс
f_akt_obn_i _otr	3 0,141435 4 0,101342 5 0,072615 6 0,052031 7 0,037282 8 0,026714 9 0,019141 10 0,013715 11 0,009827 12 0,007042 13 0,003849 14 0,002758 15 0,001976 16 0,001416	2.622585	6.86476	7.149001	8.55621
f_pass_obn _i_otr	3 0,033414 4 0,023942 5 0,017155 6 0,012292 7 0,008808 8 0,006311 9 0,004522 10 0,00324 11 0,002322 12 0,001664 13 0,000679 14 0,000487 15 0,000349 16 0,00025	0.611159	3.250196	4.375782	2.605386
f_akt_obn_i _neotr	3 0,000258 4 0,000185 5 0,000133 6 0,000095 7 0,000068 8 0,000049 9 0,000035 10 0,000025 11 0,000018 12 0,000013	0.000088	0.000000 006	1.050502	2.854634
f_pass_obn _i_neotr	4 0,012266 5 0,008789 6 0,006298 7 0,004512 8 0,003233 9 0,002317 10 0,00166 11 0,001189 12 0,000852 13 0,000611 14 0,000438 15 0,000314 16 0,000225	0.003285	0.000013	1.703639	3.579653

Продолжение табл. П.3.2.

f_neobn_i_neotr	3 4 5 6 7 8 9 10 11 12	0,101962 0,073059 0,052349 0,03751 0,026877 0,019258 0,013799 0,009887 0,007085 0,005076	0.034686	0.000931	1.050502	2.854634
-----------------	---	---	----------	----------	----------	----------

Таблица П3.3.  
Результаты работы программы для эксперимента 2

Результат работы программы	Плотность распределения вероятности времени выполнения программы запиты информации	Математическое ожидание	Дисперсия	Асимметрия	Эксцесс	
f_akt_obn_i_otr	3 4 5 6 7 8	0.024527 0.014877 0.009023 0.005473 0.002425 0.001471	0.009633	0.000064	0.632487	2.316812
f_pass_obn_i_otr	3 4 5 6 7 8	0.005994 0.003636 0.002205 0.001337 0.000428 0.00026	0.00231	4.005632 E-6	0.564136	2.269015
f_akt_obn_i_neotr	3 4 5 6	0.000045 0.000027 0.000016 9.963329e-6	0.000025	1.723214 E-10	0.251605	1.795854
f_pass_obn_i_neotr	4 5 6 7 8	0.001151 0.000698 0.000423 0.000257 0.000156	0.000537	1.279517 E-7	0.46927	2.080071
f_neobn_i_neotr	3 4 5 6	0.029696 0.018011 0.010924 0.006626	0.016314	0.000076	0.251605	1.795854
f_akt_obn_i_neotr	3 4 5 6	0.000045 0.000027 0.000016 9.963329e-6	0.000025	1.723214 E-10	0.251605	1.795854

Таблица П3.4.

## Результаты работы программы для эксперимента 3

Результат работы программы	Плотность распределения вероятности времени выполнения программы защиты информации	Математическое ожидание	Дисперсия	Асимметрия	Эксцесс
f_akt_obn_i _otr	3 0.175063 4 0.175063 5 0.175063 6 0.175063 7 0.175063 8 0.175063 9 0.175063 10 0.175063 11 0.175063 12 0.175063 13 0.106463 14 0.106463 15 0.106463 16 0.106463	0.155463	0.00096	0.9	1.9
f_pass_obn _i_otr	3 0.048188 4 0.048188 5 0.048188 6 0.048188 7 0.048188 8 0.048188 9 0.048188 10 0.048188 11 0.048188 12 0.048188 13 0.018788 14 0.018788 15 0.018788 16 0.018788	0.039787	0.000176	0.9	1.9
f_akt_obn_i _neotr	3 0.001608 4 0.001608 5 0.001608	0.001608	0		
	6 0.001608 7 0.001608 8 0.001608 9 0.001608 10 0.001608 11 0.001608 12 0.001608				
f_pass_obn _i_neotr	4 0.023734 5 0.023734 6 0.023734 7 0.023734 8 0.023734 9 0.023734 10 0.023734 11 0.023734 12 0.023734 13 0.023734 14 0.023734 15 0.023734 16 0.023734	0.023734	0		

Продолжение табл. П.3.4.

f_neobn_i_neotr	3 4 5 6 7 8 9 10 11 12	0.144219 0.144219 0.144219 0.144219 0.144219 0.144219 0.144219 0.144219 0.144219 0.144219	0.144219	0		
-----------------	---	--	----------	---	--	--

Таблица П.3.5.

Результаты работы программы для эксперимента 4

Результат работы программы	Плотность распределения вероятности времени выполнения программы защиты информации	Математическое ожидание	Дисперсия	Асимметрия	Экспесс	
f_akt_obn_i_otr	3 4 5 6 7 8	0 0 0 0.1576 0 0.0272	1.1632	4.958372	2.818005	3.976948
f_pass_obn_i_otr	3 4 5 6 7 8	0 0 0 0.0504 0 0.0048	0.3408	1.895722	3.07744	4.133502
f_akt_obn_i_neotr	3 4 5 6	0 0 0 0.0012	0.0072	0.043096	1.333333	2.333333
f_pass_obn_i_neotr	4 5 6 7 8	0 0 0 0 0.0112	0.0896	0.700834	2.25	3.25
f_neobn_i_neotr	3 4 5 6	0 0 0 0.26	1.56	3.125536	3.846154	3.846154

О правильности получаемых распределений на выходе можно судить и по внешнему виду получаемых функций. На рис. П3.2 показан график, полученный в результате обработки данных визуальным интерфейсом, в процессе эксперимента 1, и на рис. П3.3 график, полученный в пакете MathCad.

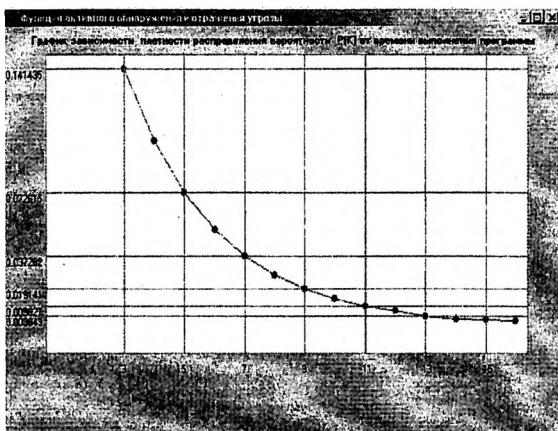


Рис. П3.2. Плотность распределения вероятности активного обнаружения и отражения угрозы по результатам работы программы с использованием MPI

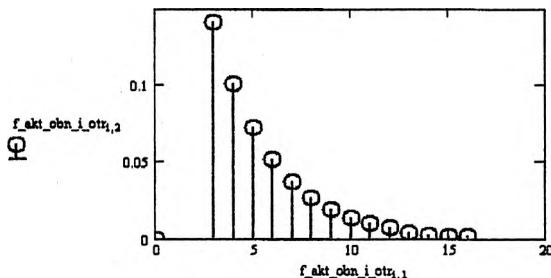


Рис. П3.3. Плотность распределения вероятности активного обнаружения и отражения угрозы по результатам работы программы с использованием MathCad

Временные затраты на выполнение программы в зависимости от размера входных данных показаны на рис.П3.4. Под размерностью входных данных подразумевается максимальное время работы подпроцессов графа, одинаковое для всех подпроцессов.



Рис.П3.4. Зависимость времени работы программы от размерности входных данных

## Текст программы

Р.П.68145-06.12.01

Ниже представлен текст главного модуля проекта по оценке динамических характеристик, разработанный на языке С с использованием библиотеки коммуникационных функций MPI, и текст заголовочного файла к проекту.

Главный модуль:

```
#include<mpi.h>
#include<stdlib.h>
#include<stdio.h>
#include"calculation.h"
#include<string.h>

//Макрос нахождения минимального из двух чисел
#define min(a, b) (((a) < (b)) ? (a) : (b))
//Макрос нахождения максимального из двух чисел
#define max(a, b) (((a) > (b)) ? (a) : (b))

ls    *start,           //указатель на начало ЛС, хранящего f(Kmax)
      *function,        //указатель на начало ЛС, хранящего f()
      *cur,             //указатель на текущий элемент ЛС
      *kduga,           //указатель на ЛС, хранящий значения f(Kduga)
      *qres,            //указатель на ЛС, хранящий значения q(Kduga)
      *temp;            //указатель на временный элемент ЛС

int FlagOrder=0;//Тип функции распределения
                  //0-экспоненциальное распределение
                  //1-равномерное распределение
                  //2-детерминированное распределение

void main (int argc, char *argv[])
{
    //Переменные

    int      ProcNum, //Число запущенных параллельных процессов
            CurID;   //Идентификатор выполняемого параллельного
//процесса

    FILE   *f_in,*f_out; //Файловые потоки ввода/вывода
    int     kmax[28];   //Массив максимальных значений времени
//выполнения блоков
    float  p[10];       //Массив вероятностей переходов между блоками в
//виде:

    //pid1,pid2,pint1,pint2,preg1,preg2,preact1,preact2,preact3,preact4;
    int     i,j;         //счетчики циклов
    double  density_sum=0;
    char   CpuName[MPI_MAX_PROCESSOR_NAME]; //Массив идентификаторов (имен)
//процессоров
    int     CpuNameLen;          //Размерность
//имен процессоров
    char   *ctemp;             //временная строка для внутренних обработчиков
    int    cursize=0;
```

```

MPI_Datatype Mpi_m_interval; //Определение дополнительного типа данных
//MPI
    m_interval k_tilda[11]; //Массив всех интервалов K-
    m_interval k_duga[10]; //массив всех интервалов Kduga
    float * mult[10]; //Массив указателей на массивы произведений,
//используемых при вычислении результатов
    float *s[4]; //Массив указателей на массивы сумм, используемых при
//вычислении результатов
    float *f_akt_chn_i_otr; //массив результатов функции активного обнаружения
//и отражения угрозы
    int delta; //размерность массива результатов
//Код

//Инициализация указателей на начало всех ЛС

start=NULL;
function=NULL;
kduga=NULL;
qres=NULL;

//Инициализация массива Kmax на основании аргументов командной строки
for (i=2;i<30;i++) kmax[i-2]=atoi(argv[i]);
//Инициализация массива P на основании аргументов командной строки
for (i=0;i<10;i++) p[i]=atof(argv[i+29+i]);
//Инициализация вида распределения FlagOrder на основании аргумента
//командной строк
FlagOrder=atoi(argv[40]);

MPI_Init(&argc, &argv); //Инициализация библиотеки MPI
MPI_Comm_size(MPI_COMM_WORLD, &ProcNum); //Получение числа процессов
MPI_Comm_rank(MPI_COMM_WORLD, &CurID); //Получение идентификатора
//текущего процесса
MPI_Get_processor_name(CpuName, &CpuNameLen); //Получение имени процессора
MPI_Type_contiguous(2, MPI_INT, &Mpi_m_interval); //Определение
//дополнительного типа MPI

//Вычисление k_tilda процессом №0
if (CurID==0) calculate_k_tilda(kmax,k_tilda);
//Мирковещательная рассыпка вычисленных значений всем процессам
MPI_Bcast(k_tilda,11,Mpi_m_interval,0,MPI_COMM_WORLD);

//Вычисление f[k]
int k=0;
for (i=0;i<28;i++) //Для каждого из 28 блоков
{
    cur=add(&start); //Добавляем элемент ЛС
    cur->array=new float[kmax[i]]; //Создаем в нем массив
//необходимого размера
    k=0; //Инициализируем K
    for (j=CurID+1;j<=kmax[i];j+=ProcNum) //Параллельно вычисляем все
//F(K)
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[j-1]=calculate_f(j,kmax[i],FlagOrder);
        //В остальных процессах заполняем все элементы подряд
        else (cur->array[k]=calculate_f(j,kmax[i],FlagOrder)); k++;
    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
snd_rcv(cur,k,ProcNum,CurID);
}

```

```

//Рассылаем результат из процесса 0 всем процессам
synchronize(28,kmax,start,CurID);

//Создание линейного списка f(ktilde)
for (int num=0;num<10;num++) //Для каждого из 10 ktilde
{
    cur=add(&function); //Добавляем элемент ЛС
    cur->array=new float[k_tilda[num].max-k_tilda[num].min+1]; //Создаем
//в нем массив необходимого размера
    k=0;
    for
(i=k_tilda[num].min+CurID;i<=k_tilda[num].max;i+=ProcNum) //Параллельно вычисляем
    все F(Ktilde)
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_tilda[num].min]=calculate_fi(num+1,i,k_tilda[num],kmax,p,FlagOrder,k_tilda);
        //В остальных процессах заполняем все элементы подряд
        else {cur-
>array[k]=calculate_fi(num+1,i,k_tilda[num],kmax,p,FlagOrder,k_tilda);k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
    send_xov(cur,k,ProcNum,CurID);

}

//Рассылаем результат из процесса 0 всем процессам
synchronize(10,k_tilda,function,CurID);
//Создание последнего элемента ЛС для Ktilde#11
cur=add(&function); //Добавляем последний 11 элемент ЛС
cur->array=new float[kmax[1]]; //Выделяем место для массива
//Выполняем вычисления для заполнения массива значений
for (i=1;i<=kmax[5];i++) cur->array[i-
1]=((calculate_fi(11,i,k_tilda[num],kmax,p,FlagOrder,k_tilda)));
//Создание ЛС f(ktilde) завершено

//Формируем массив интервалов Kduga
//(вычисляем максимальные и минимальные значения для всех Kduga)
calculate_k_duga(1,3,7,k_duga,k_tilda);
calculate_k_duga(2,1,7,k_duga,k_tilda);
calculate_k_duga(3,1,8,k_duga,k_tilda);
calculate_k_duga(4,2,7,k_duga,k_tilda);
calculate_k_duga(5,2,8,k_duga,k_tilda);
calculate_k_duga(6,3,8,k_duga,k_tilda);
calculate_k_dugaor(k_duga,k_tilda);
calculate_k_duga(8,5,10,k_duga,k_tilda);
calculate_k_duga(9,3,11,k_duga,k_tilda);
calculate_k_duga(10,6,11,k_duga,k_tilda);

//Вычисляем функции от всех k_duga - f(k_duga)
//(20 значений функций, используемых при вычислении Q(k_duga))
k=0; //Предварительная инициализация
cur=add(&kduga); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[0].max-k_duga[0].min+1];
for (i=k_duga[0].min+CurID;i<=k_duga[0].max;i+=ProcNum) //Параллельно для
//всех значений Kduga#
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[0].min]=calculate_fi(3,i,k_duga[0],kmax,p,FlagOrder,k_tilda);
    //В остальных процессах заполняем все элементы подряд
    else {cur-
>array[k]=calculate_fi(3,i,k_duga[0],kmax,p,FlagOrder,k_tilda);k++;}
}

```

```

//Заполняем в процессе 0 недостающие значения в массиве из других
//процессов
    snd_rcv(cur,k,ProcNum,CurID);
    k=0; //Предварительная инициализация
    cur->add(&kduga); //Добавляем элемент ЛС
    //Выделяем память для массива вычисляемых значений
    cur->array=new float[k_duga[1].max-k_duga[1].min+1];
    for (i=k_duga[1].min+CurID;i<=k_duga[1].max;i+=ProcNum) //Параллельно
//для всех значений Kduga#i
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_duga[1].min]=calculate_fi(1,i,k_duga[1],kmax,p,FlagOrder,k_tilda);
        //В остальных процессах заполняем все элементы подряд
        else {cur-
>array[k]=calculate_fi(1,i,k_duga[1],kmax,p,FlagOrder,k_tilda);k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
    snd_rcv(cur,k,ProcNum,CurID);
    k=0; //Предварительная инициализация
    cur->add(&kduga); //Добавляем элемент ЛС
    //Выделяем память для массива вычисляемых значений
    cur->array=new float[k_duga[2].max-k_duga[2].min+1];
    for (i=k_duga[2].min+CurID;i<=k_duga[2].max;i+=ProcNum) //Параллельно
//для всех значений Kduga#i
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_duga[2].min]=calculate_fi(1,i,k_duga[2],kmax,p,FlagOrder,k_tilda);
        //В остальных процессах заполняем все элементы подряд
        else {cur-
>array[k]=calculate_fi(1,i,k_duga[2],kmax,p,FlagOrder,k_tilda);k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
    snd_rcv(cur,k,ProcNum,CurID);
    k=0; //Предварительная инициализация
    cur->add(&kduga); //Добавляем элемент ЛС
    //Выделяем память для массива вычисляемых значений
    cur->array=new float[k_duga[3].max-k_duga[3].min+1];
    for (i=k_duga[3].min+CurID;i<=k_duga[3].max;i+=ProcNum) //Параллельно
//для всех значений Kduga#i
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_duga[3].min]=calculate_fi(2,i,k_duga[3],kmax,p,FlagOrder,k_tilda);
        //В остальных процессах заполняем все элементы подряд
        else {cur-
>array[k]=calculate_fi(2,i,k_duga[3],kmax,p,FlagOrder,k_tilda);k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
    snd_rcv(cur,k,ProcNum,CurID);
    k=0; //Предварительная инициализация
    cur->add(&kduga); //Добавляем элемент ЛС
    //Выделяем память для массива вычисляемых значений
    cur->array=new float[k_duga[4].max-k_duga[4].min+1];
    for (i=k_duga[4].min+CurID;i<=k_duga[4].max;i+=ProcNum) //Параллельно для
//всех значений Kduga#i
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_duga[4].min]=calculate_fi(2,i,k_duga[4],kmax,p,FlagOrder,k_tilda);
        //В остальных процессах заполняем все элементы подряд
        else {cur-
>array[k]=calculate_fi(2,i,k_duga[4],kmax,p,FlagOrder,k_tilda);k++;}
    }

```

```

    }

//Заполняем в процессе 0 недостающие значения в массиве из других процессов
snd_rcv(cur,k,ProcNum,CurID);
k=0; //Предварительная инициализация
cur=add(&kduga); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[5].max-k_duga[5].min+1];
for (i=k_duga[5].min+CurID;i<=k_duga[5].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[5].min]=calculate_fi(3,i,k_duga[5],kmax,p,FlagOrder,k_tilda);
    //В остальных процессах заполняем все элементы подряд
    else {cur-
>array[k]=calculate_fi(3,i,k_duga[5],kmax,p,FlagOrder,k_tilda);k++;}
}

//Заполняем в процессе 0 недостающие значения в массиве из других процессов
snd_rcv(cur,k,ProcNum,CurID);
k=0; //Предварительная инициализация
cur=add(&kduga); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[6].max-k_duga[6].min+1];
for (i=k_duga[6].min+CurID;i<=k_duga[6].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[6].min]=calculate_fi(4,i,k_duga[6],kmax,p,FlagOrder,k_tilda);
    //В остальных процессах заполняем все элементы подряд
    else {cur-
>array[k]=calculate_fi(4,i,k_duga[6],kmax,p,FlagOrder,k_tilda);k++;}
}

//Заполняем в процессе 0 недостающие значения в массиве из других процессов
snd_rcv(cur,k,ProcNum,CurID);
k=0; //Предварительная инициализация
cur=add(&kduga); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[7].max-k_duga[7].min+1];
for (i=k_duga[7].min+CurID;i<=k_duga[7].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[7].min]=calculate_fi(5,i,k_duga[7],kmax,p,FlagOrder,k_tilda);
    //В остальных процессах заполняем все элементы подряд
    else {cur-
>array[k]=calculate_fi(5,i,k_duga[7],kmax,p,FlagOrder,k_tilda);k++;}
}

//Заполняем в процессе 0 недостающие значения в массиве из других процессов
snd_rcv(cur,k,ProcNum,CurID);
k=0; //Предварительная инициализация
cur=add(&kduga); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[8].max-k_duga[8].min+1];
for (i=k_duga[8].min+CurID;i<=k_duga[8].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[8].min]=calculate_fi(3,i,k_duga[8],kmax,p,FlagOrder,k_tilda);
    //В остальных процессах заполняем все элементы подряд
}

```

```

    else {cur-
>array[k]=calculate_fi(3,i,k_duga[8],kmax,p,FlagOrder,k_tilda);k++;}
}
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
and_rcv(cur,k_ProcNum,CurID);
k=0; //Предварительная инициализация
cur=add(&kduga); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[9].max-k_duga[9].min+1];
for (i=k_duga[9].min+CurID;i<=k_duga[9].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[9].min]=calculate_fi(6,i,k_duga[9],kmax,p,FlagOrder,k_tilda);
    //В остальных процессах заполняем все элементы подряд
    else {cur-
>array[k]=calculate_fi(6,i,k_duga[9],kmax,p,FlagOrder,k_tilda);k++;}
}
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
and_rcv(cur,k_ProcNum,CurID);
//Рассылаем результат из процесса 0 всем процессам
synchronize(10,k_duga,kduga,CurID);

//Вычисляем оставшиеся 10 значений f(k_duga) для правых частей q(Kduga)

k=0; //Предварительная инициализация
cur=add(&kduga); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[0].max-k_duga[0].min+1];
for (i=k_duga[0].min+CurID;i<=k_duga[0].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[0].min]=calculate_fi(7,i,k_duga[0],kmax,p,FlagOrder,k_tilda);
    //В остальных процессах заполняем все элементы подряд
    else {cur-
>array[k]=calculate_fi(7,i,k_duga[0],kmax,p,FlagOrder,k_tilda); k++;}
}
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
and_rcv(cur,k_ProcNum,CurID);
k=0; //Предварительная инициализация
cur=add(&kduga); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[1].max-k_duga[1].min+1];
for (i=k_duga[1].min+CurID;i<=k_duga[1].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[1].min]=calculate_fi(7,i,k_duga[1],kmax,p,FlagOrder,k_tilda);
    //В остальных процессах заполняем все элементы подряд
    else {cur-
>array[k]=calculate_fi(7,i,k_duga[1],kmax,p,FlagOrder,k_tilda); k++;}
}
//Заполняем в процессе 0 недостающие значения в массиве из других процессов

and_rcv(cur,k_ProcNum,CurID);
k=0; //Предварительная инициализация
cur=add(&kduga); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[2].max-k_duga[2].min+1];

```

```

        for (i=k_duga[2].min+CurID;i<=k_duga[2].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_duga[2].min]=calculate_fi(8,i,k_duga[2],kmax,p,FlagOrder,k_tilda);
        //В остальных процессах заполняем все элементы подряд
        else {cur-
>array[k]=calculate_fi(8,i,k_duga[2],kmax,p,FlagOrder,k_tilda); k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов

        snd_rcv(cur,k,ProcNum,CurID);
        k=0; //Предварительная инициализация
        cur=>add(6kduga); //Добавляем элемент ЛС
        //Выделяем память для массива вычисляемых значений
        cur->array=new float[k_duga[3].max-k_duga[3].min+1];
        for (i=k_duga[3].min+CurID;i<=k_duga[3].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_duga[3].min]=calculate_fi(7,i,k_duga[3],kmax,p,FlagOrder,k_tilda);
        //В остальных процессах заполняем все элементы подряд
        else {cur-
>array[k]=calculate_fi(7,i,k_duga[3],kmax,p,FlagOrder,k_tilda); k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов

        snd_rcv(cur,k,ProcNum,CurID);
        k=0; //Предварительная инициализация
        cur=>add(6kduga); //Добавляем элемент ЛС
        //Выделяем память для массива вычисляемых значений
        cur->array=new float[k_duga[4].max-k_duga[4].min+1];
        for (i=k_duga[4].min+CurID;i<=k_duga[4].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_duga[4].min]=calculate_fi(8,i,k_duga[4],kmax,p,FlagOrder,k_tilda);
        //В остальных процессах заполняем все элементы подряд
        else {cur-
>array[k]=calculate_fi(8,i,k_duga[4],kmax,p,FlagOrder,k_tilda); k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов

        snd_rcv(cur,k,ProcNum,CurID);
        k=0; //Предварительная инициализация
        cur=>add(6kduga); //Добавляем элемент ЛС
        //Выделяем память для массива вычисляемых значений
        cur->array=new float[k_duga[5].max-k_duga[5].min+1];
        for (i=k_duga[5].min+CurID;i<=k_duga[5].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_duga[5].min]=calculate_fi(8,i,k_duga[5],kmax,p,FlagOrder,k_tilda);
        //В остальных процессах заполняем все элементы подряд
        else {cur-
>array[k]=calculate_fi(8,i,k_duga[5],kmax,p,FlagOrder,k_tilda); k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов

        snd_rcv(cur,k,ProcNum,CurID);

```

```

k=0; //Предварительная инициализация
cur=add(&kduga); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[6].max-k_duga[6].min+1];
for (i=k_duga[6].min+CurID;i<=k_duga[6].max;i+=ProcNum) //Параллельно для
//всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[6].min]=calculate_fi(9,i,k_duga[6],kmax,p,FlagOrder,k_tilda);
    //В остальных процессах заполняем все элементы подряд
    else {cur-
>array[i]=calculate_fi(9,i,k_duga[6],kmax,p,FlagOrder,k_tilda);k++;}
}
//Заполняем в процессе 0 недостающие значения в массиве из других процессов

snd_rcv(cur,k,ProcNum,CurID);
k=0; //Предварительная инициализация
cur=add(&kduga); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[7].max-k_duga[7].min+1];
for (i=k_duga[7].min+CurID;i<=k_duga[7].max;i+=ProcNum) //Параллельно для
//всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[7].min]=calculate_fi(10,i,k_duga[7],kmax,p,FlagOrder,k_tilda);
    //В остальных процессах заполняем все элементы подряд
    else {cur-
>array[i]=calculate_fi(10,i,k_duga[7],kmax,p,FlagOrder,k_tilda);k++;}
}
//Заполняем в процессе 0 недостающие значения в массиве из других процессов

snd_rcv(cur,k,ProcNum,CurID);
k=0; //Предварительная инициализация
cur=add(&kduga); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[8].max-k_duga[8].min+1];
for (i=k_duga[8].min+CurID;i<=k_duga[8].max;i+=ProcNum) //Параллельно для
//всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[8].min]=calculate_f11_q(i,k_duga[8],kmax,p,FlagOrder);
    //В остальных процессах заполняем все элементы подряд
    else {cur-
>array[i]=calculate_f11_q(i,k_duga[8],kmax,p,FlagOrder);k++;}
}
//Заполняем в процессе 0 недостающие значения в массиве из других процессов

snd_rcv(cur,k,ProcNum,CurID);
k=0; //Предварительная инициализация
cur=add(&kduga); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[9].max-k_duga[9].min+1];
for (i=k_duga[9].min+CurID;i<=k_duga[9].max;i+=ProcNum) //Параллельно для
//всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[9].min]=calculate_f11_q(i,k_duga[9],kmax,p,FlagOrder);
    //В остальных процессах заполняем все элементы подряд
    else {cur-
>array[i]=calculate_f11_q(i,k_duga[9],kmax,p,FlagOrder);k++;}
}

```

```

    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов

    snd_rcv(curl,k,ProcNum,CurID);

//Устанавливаем указатель на 10 элемент ЛС для рассылки во все процессы только
//последних 10 элементов
    temp=k_duga;
    for (i=0;i<10;i++) temp=temp->next;
//Рассылаем значения из 0 процесса всем остальным процессам
    synchronize(10,k_duga,temp,CurID);

//Вычисление значений Q(k_duga)
    k=0; //Предварительная инициализация
    cur=add(&qres); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
    cur->array=new float[k_duga[0].max-k_duga[0].min+1];
    for (i=k_duga[0].min+CurID;i<=k_duga[0].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_duga[0].min]=calculate_qi(1,i,7,1,3,k_tilda,k_duga);
        //В остальных процессах заполняем все элементы подряд
        else {cur->array[k]=calculate_qi(1,i,7,1,3,k_tilda,k_duga);k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других
//процессов
    snd_rcv(curl,k,ProcNum,CurID);
    k=0; //Предварительная инициализация
    cur=add(&qres); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
    cur->array=new float[k_duga[1].max-k_duga[1].min+1];
    for (i=k_duga[1].min+CurID;i<=k_duga[1].max;i+=ProcNum) //Параллельно
//для всех значений Kduga#i
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_duga[1].min]=calculate_qi(2,i,7,2,1,k_tilda,k_duga);
        //В остальных процессах заполняем все элементы подряд
        else {cur->array[k]=calculate_qi(2,i,7,2,1,k_tilda,k_duga);k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
    snd_rcv(curl,k,ProcNum,CurID);
    k=0; //Предварительная инициализация
    cur=add(&qres); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
    cur->array=new float[k_duga[2].max-k_duga[2].min+1];
    for (i=k_duga[2].min+CurID;i<=k_duga[2].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_duga[2].min]=calculate_qi(3,i,8,3,1,k_tilda,k_duga);
        //В остальных процессах заполняем все элементы подряд
        else {cur->array[k]=calculate_qi(3,i,8,3,1,k_tilda,k_duga);k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других
//процессов
    snd_rcv(curl,k,ProcNum,CurID);
    k=0; //Предварительная инициализация
    cur=add(&qres); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
    cur->array=new float[k_duga[3].max-k_duga[3].min+1];

```

```

    for (i=k_duga[3].min+CurID;i<=k_duga[3].max;i+=ProcNum) //Параллельно
//для всех значений Kduga#i
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_duga[3].min]=calculate_qi(4,i,7,4,2,k_tilda,k_duga);
        //В остальных процессах заполняем все элементы подряд
        else (cur->array[k]=calculate_qi(4,i,7,4,2,k_tilda,k_duga));k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
snd_rcv(cur,k,ProcNum,CurID);
k=0; //Предварительная инициализация
cur=add(&qres); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[4].max-k_duga[4].min+1];
for (i=k_duga[4].min+CurID;i<=k_duga[4].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[4].min]=calculate_qi(5,i,8,5,2,k_tilda,k_duga);
    //В остальных процессах заполняем все элементы подряд
    else (cur->array[k]=calculate_qi(5,i,8,5,2,k_tilda,k_duga));k++;}
}
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
snd_rcv(cur,k,ProcNum,CurID);
k=0; //Предварительная инициализация
cur=add(&qres); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[5].max-k_duga[5].min+1];
for (i=k_duga[5].min+CurID;i<=k_duga[5].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[5].min]=calculate_qi(6,i,8,6,3,k_tilda,k_duga);
    //В остальных процессах заполняем все элементы подряд
    else (cur->array[k]=calculate_qi(6,i,8,6,3,k_tilda,k_duga));k++;}
}
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
snd_rcv(cur,k,ProcNum,CurID);
k=0; //Предварительная инициализация
cur=add(&qres); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[6].max-k_duga[6].min+1];
for (i=k_duga[6].min+CurID;i<=k_duga[6].max;i+=ProcNum)//Параллельно для
//всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum
    if (CurID==0) cur->array[i-
k_duga[6].min]=calculate_qi_or(7,i,9,7,4,k_tilda,k_duga);
    //В остальных процессах заполняем все элементы подряд
    else (cur->array[k]=calculate_qi_or(7,i,9,7,4,k_tilda,k_duga));k++;}
}
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
snd_rcv(cur,k,ProcNum,CurID);
k=0; //Предварительная инициализация
cur=add(&qres); //Добавляем элемент ЛС
//Выделяем память для массива вычисляемых значений
cur->array=new float[k_duga[7].max-k_duga[7].min+1];
for (i=k_duga[7].min+CurID;i<=k_duga[7].max;i+=ProcNum) //Параллельно
//для всех значений Kduga#i
{
    //В процессе 0 заполняем массив с шагом step=ProcNum

```

```

        if (CurID==0) cur->array[i-
k_duga[7].min]=calculate_qi(8,i,10,8,5,k_tilda,k_duga);
        //В остальных процессах заполняем все элементы подряд
        else {cur->array[k]=calculate_qi(8,i,10,8,5,k_tilda,k_duga);k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
    snd_rcv(cur,k,ProcNum,CurID);
    k=0; //Предварительная инициализация
    cur=add(&qres); //Добавляем элемент ЛС
    //Выделяем память для массива вычисляемых значений
    cur->array=new float[k_duga[8].max-k_duga[8].min+1];
    for (i=k_duga[8].min+CurID;i<=k_duga[8].max;i+=ProcNum) //Параллельно
//для всех значений Kduga#i
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_duga[8].min]=calculate_qi(9,i,11,9,3,k_tilda,k_duga);
        //В остальных процессах заполняем все элементы подряд
        else {cur->array[k]=calculate_qi(9,i,11,9,3,k_tilda,k_duga);k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
    snd_rcv(cur,k,ProcNum,CurID);
    k=0; //Предварительная инициализация
    cur=add(&qres); //Добавляем элемент ЛС
    //Выделяем память для массива вычисляемых значений
    cur->array=new float[k_duga[9].max-k_duga[9].min+1];
    for (i=k_duga[9].min+CurID;i<=k_duga[9].max;i+=ProcNum) //Параллельно
//для всех значений Kduga#i
    {
        //В процессе 0 заполняем массив с шагом step=ProcNum
        if (CurID==0) cur->array[i-
k_duga[9].min]=calculate_qi(10,i,11,10,6,k_tilda,k_duga);
        //В остальных процессах заполняем все элементы подряд
        else {cur->array[k]=calculate_qi(10,i,11,10,6,k_tilda,k_duga);k++;}
    }
//Заполняем в процессе 0 недостающие значения в массиве из других процессов
    snd_rcv(cur,k,ProcNum,CurID);
    //Передача корректных значений из процесса 0 всем остальным процессам
    synchronise(10,k_duga,qres,CurID);

//Выполняем в процессе 0 вычисление окончательных результатов и запись их в файл
    //для их последующей обработки
    if (CurID==0)
    {
        //Выполняем вычисление всех произведений и заполнение массива результатов
        for (i=0;i<10;i++) mult[i]=calculate_mult(i+1,i+19,k_duga,kmax);
        //Выполняем все необходимые суммирования
        s[0]=calculate_sum(mult[0],mult[1],k_duga[0].min+1,k_duga[0].max+kmax[18],
k_duga[1].min+1,k_duga[1].max+kmax[19]);
        s[1]=calculate_sum(mult[2],mult[3],k_duga[2].min+1,k_duga[2].max+kmax[20],
k_duga[3].min+1,k_duga[3].max+kmax[21]);
        s[2]=calculate_sum(mult[4],mult[5],k_duga[4].min+1,k_duga[4].max+kmax[22],
k_duga[5].min+1,k_duga[5].max+kmax[23]);
        s[3]=calculate_sum(mult[6],mult[9],k_duga[8].min+1,k_duga[8].max+kmax[26],
k_duga[9].min+1,k_duga[9].max+kmax[27]);
        //Формируем массив результатов для функции активного обнаружения и отражения
        //уровни
        f_akt_obn_i_otr=calculate_sum(s[0],s[1],
            min(k_duga[0].min+1,k_duga[1].min+1),max(k_duga[0].max+kmax[18],k_duga[1].
max+kmax[19]),
            min(k_duga[2].min+1,k_duga[3].min+1),max(k_duga[2].max+kmax[20],k_duga[3].
max+kmax[21]));
    }

```

```

    //Вычисляем интервал результатов для функции активного обнаружения и
    //отражения угрозы
    facio.min=min(min(k_duga[0].min+1,k_duga[1].min+1),min(k_duga[2].min+1,k_d
    uga[3].min+1));
    facio.max=max(max(k_duga[0].max+kmax[18],k_duga[1].max+kmax[19]),max(k_dug
    a[2].max+kmax[20],k_duga[3].max+kmax[21]));
    //Вычисляем МО для функции активного обнаружения и отражения угрозы
    mo=stat_mo(facio.min,facio.max,f_akt_obn_i_otr);
    //Открываем (создаем) файл результатов
    f_out=fopen("1.res","wt");
    //Выводим в файл число результатов
    fprintf(f_out,"%d ",facio.max-facio.min+1);
    //Выводим в файл значения времен
    for (i=facio.min;i<=facio.max;i++)
        fprintf(f_out,"%d ",i);
    fprintf(f_out,"\n");
    //Выводим в файл значения функции плотности распределения вероятности
    //времен выполнения программы
    for (i=0;i<facio.max-facio.min+1;i++)
        fprintf(f_out, "%f ",f_akt_obn_i_otr[i]);
    //Выводим в файл значение математического ожидания
    fprintf(f_out,"%f",mo);
    //Закрываем файл
    fclose(f_out);

    //Вычисляем количество результатов для функции пассивного обнаружения и
    //отражения угрозы
    delta = (max((k_duga[4].max+kmax[22]),(k_duga[5].max+kmax[23]))-
    min((k_duga[4].min+1),(k_duga[5].min+1)));
    //Вычисляем МО для функции пассивного обнаружения и отражения угрозы
    mo=stat_mo(min((k_duga[4].min+1),(k_duga[5].min+1)),max((k_duga[4].max+kma
    x[22]),(k_duga[5].max+kmax[23])),s[2]);
    //Открываем (создаем) файл результатов
    f_out=fopen("2.res","wt");
    //Выводим в файл число результатов
    fprintf(f_out,"%d ",delta+1);
    //Выводим в файл значения времен
    for
    (i=min((k_duga[4].min+1),(k_duga[5].min+1));i<=max((k_duga[4].max+kmax[22]),(k_d
    uga[5].max+kmax[23]));i++)
        fprintf(f_out,"%d ",i);
    fprintf(f_out,"\n");
    //Выводим в файл значения функции плотности распределения вероятности
    //времен выполнения программы
    for
    (i=min((k_duga[4].min+1),(k_duga[5].min+1));i<=max((k_duga[4].max+kmax[22]),(k_d
    uga[5].max+kmax[23]));i++)
        fprintf(f_out, "%f ",s[2][i-
    min((k_duga[4].min+1),(k_duga[5].min+1))]);
    //Выводим в файл значение математического ожидания
    fprintf(f_out,"%f",mo);
    //Закрываем файл
    fclose(f_out);

    //Вычисляем количество результатов для функции активного обнаружения и
    //неотражения угрозы
    delta =k_duga[6].max+kmax[24]-k_duga[6].min-1;
    //Вычисляем МО для функции активного обнаружения и неотражения угрозы
    mo=stat_mo(k_duga[6].min+1, facio.max, f_akt_obn_i_otr);
    //Открываем (создаем) файл результатов
    f_out=fopen("3.res","wt");
    //Выводим в файл число результатов
    fprintf(f_out,"%d ",delta+1);
    //Выводим в файл значения времен

```

```

for (i=k_duga[6].min+1;i<=k_duga[6].max+kmax[24];i++)
    fprintf(f_out,"%d ",i);
fprintf(f_out,"n");
//Выводим в файл значения функции плотности распределения вероятности
//времен выполнения программы
for (i=k_duga[6].min+1;i<=k_duga[6].max+kmax[24];i++)
    fprintf(f_out, "%f ",mult[6][i-k_duga[6].min-1]);
//Выводим в файл значение математического ожидания
fprintf(f_out,"%f",mo);
//Закрываем файл
fclose(f_out);

//Вычисляем количество результатов для функции пассивного обнаружения и
//неотражения угрозы
delta =k_duga[7].max+kmax[25]-k_duga[7].min-1;
//Вычисляем МО для функции пассивного обнаружения и неотражения угрозы
mo=stat_mo(k_duga[7].min+1,k_duga[7].max+kmax[25],mult[7]);
//Открываем (создаем) файл результатов
f_out=fopen("4.res","wt");
//Выводим в файл число результатов
fprintf(f_out,"%d ",delta+1);
//Выводим в файл значения времен
for (i=k_duga[7].min+1;i<=k_duga[7].max+kmax[25];i++)
    fprintf(f_out,"%d ",i);
fprintf(f_out,"n");
//Выводим в файл значения функции плотности распределения вероятности
//времен выполнения программы
for (i=k_duga[7].min+1;i<=k_duga[7].max+kmax[25];i++)
    fprintf(f_out, "%f ",mult[7][i-k_duga[7].min-1]);
//Выводим в файл значение математического ожидания
fprintf(f_out,"%f",mo);
//Закрываем файл
fclose(f_out);

//Вычисляем количество результатов для функции необнаружения и неотражения
//угрозы
delta =(max((k_duga[8].max+kmax[26]),(k_duga[9].max+kmax[27]))-
min((k_duga[8].min+1),(k_duga[9].min+1)));
//Вычисляем МО для функции необнаружения и неотражения угрозы
mo=stat_mo(min((k_duga[8].min+1),(k_duga[9].min+1)),max((k_duga[8].max+kma
x[26]),(k_duga[9].max+kmax[27])),s[3]);
//Открываем (создаем) файл результатов
f_out=fopen("5.res","wt");
//Выводим в файл число результатов
fprintf(f_out,"%d ",delta+1);
//Выводим в файл значения времен
for
(i=min((k_duga[8].min+1),(k_duga[9].min+1));i<=max((k_duga[8].max+kmax[26]),(k_d
uga[9].max+kmax[27]));i++)
    fprintf(f_out,"%d ",i);
fprintf(f_out,"n");
//Выводим в файл значения функции плотности распределения вероятности
//времен выполнения программы
for
(i=min((k_duga[8].min+1),(k_duga[9].min+1));i<=max((k_duga[8].max+kmax[26]),(k_d
uga[9].max+kmax[27]));i++)
    fprintf(f_out, "%f ",s[3][i]);
//Выводим в файл значение математического ожидания
fprintf(f_out,"%f",mo);
//Закрываем файл
fclose(f_out);
}
MPI_Finalize(); //Завершение работы с библиотекой MPI

```

```

}

//Добавление элемента в ЛС
ls* add(
    ls* m_start) //Указатель на адрес начала ЛС
{
    ls *temp; //Временный элемент
    ls *current;//Текущий элемент ЛС
    temp =new ls; //Создаем новый элемент ЛС
    if (temp==NULL) {printf("Elem doesn't create!!\n");} //Проверяем успешно-
//ли создан
    temp->next=NULL; //Инициализируем указатель на следующий элемент
    temp->array=NULL; //Инициализируем указатель на массив
    if (*m_start==NULL) (*m_start=temp); //Если это первый элемент,
//устанавливаем начало

    else
    {
        current=*m_start; //В противном случае переходим к последнему
//элементу
        while (current->next!=NULL)
            current=current->next; //Устанавливаем его указатель на следующий
//элемент
        current->next=temp; //на созданный элемент
    }
    return temp; //Возвращаем указатель на созданный элемент
}

//Выбор нужного элемента массива в ЛС
float run(int ix1,//индекс элемента ЛС
          int ix2,//индекс элемента массива
          ls* start)//указатель на начало ЛС
{
    ls *current; //текущий элемент ЛС
    current=start; //устанавливаем указатель в начало ЛС
    while (1) //переходим к нужному элементу ЛС
    {
        if (ix1<=0) break;
        current=current->next;
        ix1--;
    }
    return current->array[ix2-1]; //Возвращаем нужный элемент массива
}

//Функция вычисления плотности распределения времени выполнения блока
float calculate_f(int k, //время выполнения
                  int kmax, //максимальное время выполнения
                  int FlagOrder) //тип распределения
{
    if (FlagOrder==0) //Экспоненциальный тип распределения
    {
        if (kmax<=0 ) return 0; //обработка некорректного максимального
//значения
        else
        {
            float res=0;
            float med=0;
            for (int i=1;i<=kmax;i++) //Вычисление среднего значения
            {
                med+=i;
            }
        }
    }
}

```

```

med=med/kmax;
int r=ceil(med); //Округление среднего до целого
if (r-med<=-0.5) r=floor(med);
res=exp(-float(k)/r)/r; //Вычисление функции по формуле
return res; //Результат
}

}

if (FlagOrder==1) //Равномерный тип распределения
{
    float res;
    res=1./kmax;      //Результат=1/максимум
    return res;
}

if (FlagOrder==2) //Детерминированный тип распределения
{
    float res;
    if (k==sel_num) return 1.; //Получаем единицу только в одном
//случае
    else return 0.;           //в остальных - 0
}

}

//Функция вычисления всех Ktilda
void calculate_k_tilda(int array[],m_interval k_tilda[])
{
    k_tilda[0].min=2;
    if (array[0]+array[10]>array[1]+array[6])
k_tilda[0].max=array[0]+array[10];
    else k_tilda[0].max=array[1]+array[6]+array[10];
    k_tilda[1].min=2;
    if (array[0]+array[11]>array[1]+array[6]+array[11])
k_tilda[1].max=array[0]+array[11];
    else k_tilda[1].max=array[1]+array[6]+array[11];
    k_tilda[2].min=2;
    k_tilda[2].max=array[1]+array[7];
    k_tilda[3].min=3;
    k_tilda[3].max=array[2]+array[8]+array[12];
    k_tilda[4].min=3;
    k_tilda[4].max=array[2]+array[8]+array[13];
    k_tilda[5].min=2;
    k_tilda[5].max=array[2]+array[9];
    k_tilda[6].min=2;
    k_tilda[6].max=array[3]+array[14];
    k_tilda[7].min=2;
    k_tilda[7].max=array[3]+array[15];
    k_tilda[8].min=2;
    k_tilda[8].max=array[4]+array[16];
    k_tilda[9].min=2;
    k_tilda[9].max=array[4]+array[17];
    k_tilda[10].min=1;
    k_tilda[10].max=array[5];
}

//Функция f1(K)
float calculate_f1(int k_tilda,struct m_interval &k_tilda_int,int k_array[],
float p_array[],int FlagOrder,m_interval k_tilda_instance[])
{
    int i=0,j=0;
    float sum=0,sum2=0,sum3=0;
}

```

```

        for (i=1;i<=k_array[0];i++)
            sum+=run(0,i,start)*calculate_f(k_tilda-i,k_tilda_int.max-
k_array[0],FlagOrder);

        sum+=sum*p_array[0]*p_array[6];
        for (i=2;i<=k_array[1]+k_array[6];i++)
        {
            sum2=0;
            for (j=1;j<=k_array[1];j++)
                sum2+=run(1,j,start)*calculate_f(i-
j,k_array[6],FlagOrder);
            sum3+=sum2*calculate_f(k_tilda-i,k_tilda_int.max-k_array[1]-
k_array[6],FlagOrder);
        }
        sum+=sum3*p_array[1]*p_array[4]*p_array[6];
        return sum;
    }

//Функция f2(K)
float calculate_f2(int k_tilda,struct m_interval &k_tilda_int,int k_array[],
float p_array[],int FlagOrder,m_interval k_tilda_instance[])
{
    int i=0,j=0;
    float sum=0,sum2=0,sum3=0;
    for (i=1;i<=k_array[0];i++)
        sum+=run(0,i,start)*calculate_f(k_tilda-i,k_tilda_int.max-
k_array[0],FlagOrder);

    sum+=sum*p_array[0]*(1-p_array[6]);
    for (i=2;i<=k_array[1]+k_array[6];i++)
    {
        sum2=0;
        for (j=1;j<=k_array[1];j++)
            sum2+=run(1,j,start)*calculate_f(i-j,k_array[6],FlagOrder);
        sum3+=sum2*calculate_f(k_tilda-i,k_tilda_int.max-k_array[1]-
k_array[6],FlagOrder);
    }
    sum+=sum3*p_array[1]*p_array[4]*(1-p_array[6]);
    return sum;
}

//Функция f3(K)
float calculate_f3(int k_tilda,struct m_interval &k_tilda_int,int k_array[],
float p_array[],int FlagOrder,m_interval k_tilda_instance[])
{
    int i=0,j=0;
    float sum=0;
    for (i=1;i<=k_array[1];i++)
        sum+=run(1,i,start)*calculate_f(k_tilda-i,k_tilda_int.max-
k_array[1],FlagOrder);
    sum+=sum*p_array[1]*(1-p_array[4]);
    return sum;
}

//Функция f4(K)
float calculate_f4(int k_tilda,struct m_interval &k_tilda_int,int k_array[],
float p_array[],int FlagOrder,m_interval k_tilda_instance[])
{
    int i=0,j=0;
    float sum=0,sum2=0;
    for (i=2;i<=k_array[2]+k_array[8];i++)
    {
        sum=0;
        for (j=1;j<=k_array[2];j++)

```

```

        sum+=run(2,j,start)*calculate_f(i-j,k_array[8],FlagOrder);
        sum2+=sum*calculate_f(k_tilda-i,k_tilda_int.max-k_array[2]-
k_array[8],FlagOrder);
    }
    sum2+=sum*p_array[5]*p_array[7]*(1-p_array[0]-p_array[1]);
    return sum2;
}

//Функция f5(K)
float calculate_f5(int k_tilda,struct m_interval &k_tilda_int,int k_array[],
float p_array[],int FlagOrder,m_interval k_tilda_instance[])
{
    int i=0,j=0;
    float sum=0,sum2=0;
    for (i=2;i<=k_array[2]+k_array[8];i++)
    {
        sum=0;
        for (j=1;j<=k_array[2];j++)
            sum+=run(2,j,start)*calculate_f(i-j,k_array[8],FlagOrder);
        sum2+=sum*calculate_f(k_tilda-i,k_tilda_int.max-k_array[2]-
k_array[8],FlagOrder);
    }
    sum2+=sum*p_array[5]*(1-p_array[7))*(1-p_array[0]-p_array[1]);
    return sum2;
}

//Функция f6(K)
float calculate_f6(int k_tilda,struct m_interval &k_tilda_int,int k_array[],
float p_array[],int FlagOrder,m_interval k_tilda_instance[])
{
    int i=0,j=0;
    float sum=0;
    for (i=1;i<=k_array[1];i++)
        sum+=run(2,i,start)*calculate_f(k_tilda-i,k_tilda_int.max-
k_array[2],FlagOrder);
    sum+=sum*(1-p_array[1]-p_array[0])*(1-p_array[5]);
    return sum;
}

//Функция f7(K)
float calculate_f7(int k_tilda,struct m_interval &k_tilda_int,int k_array[],
float p_array[],int FlagOrder,m_interval k_tilda_instance[])
{
    int i=0,j=0;
    float sum=0;
    for (i=1;i<=k_array[3];i++)
        sum+=run(3,i,start)*calculate_f(k_tilda-i,k_tilda_int.max-
k_array[3],FlagOrder);
    sum+=sum*p_array[2]*p_array[8];
    return sum;
}

//Функция f8(K)
float calculate_f8(int k_tilda,struct m_interval &k_tilda_int,int k_array[],
float p_array[],int FlagOrder,m_interval k_tilda_instance[])
{
    int i=0,j=0;
    float sum=0;
    for (i=1;i<=k_array[3];i++)
        sum+=run(3,i,start)*calculate_f(k_tilda-i,k_tilda_int.max-
k_array[3],FlagOrder);
    sum+=sum*p_array[2]*(1-p_array[8]);
    return sum;
}

```

```

}

//Функция f9(K)
float calculate_f9(int k_tilda,struct m_interval &k_tilda_int,int k_array[],
float p_array[],int FlagOrder,m_interval k_tilda_instance[])
{
    int i=0;
    float sum=0;
    for (i=1;i<=k_array[4];i++)
        sum+=run(4,i,start)*calculate_f(k_tilda-i,k_tilda_int.max-
k_array[4],FlagOrder);///////
    sum=sum*p_array[3]*p_array[9];
    return sum;
}

//Функция f10(K)
float calculate_f10(int k_tilda,struct m_interval &k_tilda_int,int k_array[],
float p_array[],int FlagOrder,m_interval k_tilda_instance[])
{
    int i=0,j=0;
    float sum=0;
    for (i=1;i<=k_array[4];i++)
        sum+=run(4,i,start)*calculate_f(k_tilda-i,k_tilda_int.max-
k_array[4],FlagOrder);
    sum=sum*p_array[3]*(1-p_array[9]);
    return sum;
}

//Функция f11(K)
float calculate_f11(int k_tilda,struct m_interval &k_tilda_int,int k_array[],
float p_array[],int FlagOrder)
{
    float sum=0;
    sum=run(5,k_tilda,start)*(1-p_array[2]-p_array[3]);
    return sum;
}

//Функция f1(Q)
float calculate_f11_q(int k_tilda,struct m_interval &k_tilda_int,int k_array[],
float p_array[],int FlagOrder)
{
    float sum=0;
    sum=calculate_f(k_tilda,k_tilda_int.max,FlagOrder)*(1-p_array[2]-
p_array[3]);
    return sum;
}

//Функция выбора необходимой функции для вычислений в зависимости от аргумента
/int number/
float calculate_fi(int number,int k_tilda,struct m_interval &k_tilda_int,int
k_array[], float p_array[],int FlagOrder,m_interval k_tilda_instance[])
{
    //Вызывается одна из 11 функций в зависимости от аргумента number
    float res=0;
    switch (number)
    {
        case
1:(res=calculate_f1(k_tilda,k_tilda_int,k_array,p_array,FlagOrder,k_tilda_instan-
ce));
            break;
        case
2:(res=calculate_f2(k_tilda,k_tilda_int,k_array,p_array,FlagOrder,k_tilda_instan-
ce));
            break;
    }
}

```

```

    case
3:{res=calculate_f3(k_tilda,k_tilda_int,k_array,p_array,FlagOrder,k_tilda_instance);
     break;}
    case
4:{res=calculate_f4(k_tilda,k_tilda_int,k_array,p_array,FlagOrder,k_tilda_instance);
     break;}
    case
5:{res=calculate_f5(k_tilda,k_tilda_int,k_array,p_array,FlagOrder,k_tilda_instance);
     break;}
    case
6:{res=calculate_f6(k_tilda,k_tilda_int,k_array,p_array,FlagOrder,k_tilda_instance);
     break;}
    case
7:{res=calculate_f7(k_tilda,k_tilda_int,k_array,p_array,FlagOrder,k_tilda_instance);
     break;}
    case
8:{res=calculate_f8(k_tilda,k_tilda_int,k_array,p_array,FlagOrder,k_tilda_instance);
     break;}
    case
9:{res=calculate_f9(k_tilda,k_tilda_int,k_array,p_array,FlagOrder,k_tilda_instance);
     break;}
    case
10:{res=calculate_f10(k_tilda,k_tilda_int,k_array,p_array,FlagOrder,k_tilda_instance);
      break;}
    case
11:res=calculate_f11(k_tilda,k_tilda_int,k_array,p_array,FlagOrder);
}
return res; //возвращаем результат вычислений
}

//функция для вычисления K_duga для блоков, объединяемых по AND
void calculate_k_duga(int num,int ix1,int ix2,m_interval res[],m_interval k_tilda[])
{
    if (k_tilda[ix1-1].min<k_tilda[ix2-1].min) res[num-1].min=k_tilda[ix2-1].min;
    else res[num-1].min=k_tilda[ix1-1].min;
    if (k_tilda[ix1-1].max<k_tilda[ix2-1].max) res[num-1].max=k_tilda[ix2-1].max;
    else res[num-1].max=k_tilda[ix1-1].max;
}

//функция для вычисления K_duga для блоков, объединяемых по OR
void calculate_k_dugaor(m_interval res[],m_interval k_tilda[])
{
    if (k_tilda[3].min<k_tilda[8].min) res[6].min=k_tilda[3].min;
    else res[6].min=k_tilda[8].min;
    if (k_tilda[3].max<k_tilda[8].max) res[6].max=k_tilda[3].max;
    else res[6].max=k_tilda[8].max;
}

//функция вычисления Qi(Kduga) при объединении блоков по AND
//calculate_qi(           1,           i,           7,           1,           3,
k_duga)
float calculate_qi(int num_func,
                   int arg,

```

```

        int ix1,
        int ix2,
        int ix3,
        m_interval k_tilda[],
        m_interval k_duga[])
    }

    float sum2=0,sum=0;
    int i=0;
    int lim1=0,lim2=0;

    if (arg<=k_tilda[ix1-1].max) lim1=arg; else lim1=k_tilda[ix1-1].max;
    if (arg<=k_tilda[ix3-1].max) lim2=arg; else lim2=k_tilda[ix3-1].max;

    for (i=k_tilda[ix1-1].min;i<=lim1;i++)
        sum+=run(ix1-1,i-k_tilda[ix1-1].min+1,function);
    sum+=sum*run(num_func-1,arg-k_duga[num_func-1].min+1,kduga);

    for (i=k_tilda[ix3-1].min;i<lim2;i++)
        sum2+=run(ix3-1,i-k_tilda[ix3-1].min+1,function);
    sum2+=sum2*run(num_func+9,arg-k_duga[num_func-1].min+1,kduga);

    sum2+=sum;

    return sum2;
}

//Функция вычисления Qi(Kduga) при объединении блоков по OR
float calculate_qi_or(int num_func,
                      int arg,
                      int ix1,
                      int ix2,
                      int ix3,
                      m_interval k_tilda[],
                      m_interval k_duga[])
{
    float sum2=0,sum=0;
    int i=0;

    for (i=arg;i<=k_tilda[ix1-1].max;i++)
        sum+=run(ix1-1,i-k_tilda[ix1-1].min+1,function);
    sum+=sum*run(num_func-1,arg-k_duga[num_func-1].min+1,kduga);

    for (i=arg+1;i<=k_tilda[ix3-1].max;i++)
        sum2+=run(ix3-1,i-k_tilda[ix3-1].min+1,function);
    sum2+=sum2*run(num_func+9,arg-k_duga[num_func-1].min+1,kduga);

    sum2+=sum;

    return sum2;
}

//Функция вычисления произведения
float *calculate_mult(int ix1,           //индекс первого сомножителя
                      int ix2,           //индекс второго сомножителя
                      m_interval k_duga[], //интервал значений Kduga
                      int kmax[])        //максимальные времена выполнения
//блоков
{
    float sum=0;//сумма
    float *res; //указатель на массив результатов
    int i=0, j=0;//счетчики
    int lim1=0,lim2=0;//пределы обработки циклов
}

```

```

//выделяем память под массив результатов
res=new float[k_duga[ixl-1].max+kmax[ix2-1]-k_duga[ixl-1].min];
//Обнуление массива результатов
for (i=0;i<=k_duga[ixl-1].max+kmax[ix2-1]-k_duga[ixl-1].min;i++) res[i]=0;
//Для всех элементов массива результатов
for (j=k_duga[ixl-1].min+1;j<=k_duga[ixl-1].max+kmax[ix2-1];j++)
{
    //вычисляем и накапливаем произведение
    for (i=k_duga[ixl-1].min;i<=k_duga[ixl-1].max;i++)
        sum+=run(ix1-1,i-(k_duga[ix1-1].min)+1,qres)*calculate_f(j-1,kmax[ix2-1],FlagOrder);
    res[j-k_duga[ixl-1].min-1]=sum; //сохраняем результат
    sum=0; //обнуляем промежуточную сумму
}
return res; //возвращаем указатель на массив результатов
}

//Функция вычисления суммы
float * calculate_sum (float *s1, //указатель на массив первого слагаемого
                      float *s2, //указатель на массив второго
//слагаемого
                      int a, //минимальное значение первого
//слагаемого
                      int b, //максимальное значение первого
//слагаемого
                      int c, //минимальное значение второго
//слагаемого
                      int d) //максимальное значение второго
{
    float *sum;
    //Выделяем память под массив результатов
    sum=new float[max(b,d)-min(a,c)+1];
    //Для всех элементов
    for (int i=0;i<=max(b,d)-min(a,c);i++) sum[i]=0;
    //Если минимальное значение первого слагаемого меньше, чем второго
    if (a<c)
    {
        //Если максимальное значение первого слагаемого больше, чем второго
        if (b>d)
        {
            //Формируем элементы от минимального элемента первого слагаемого до
            //минимального элемента второго слагаемого как элементы первого слагаемого
            for (i=0;i<c-a;i++)
                sum[i]=s1[i];
            //Формируем элементы от минимального элемента второго слагаемого до
            //максимального элемента второго слагаемого как сумму элементов слагаемых
            for (i=c-a;i<d;i++)
                sum[i]=s1[i]+s2[i-(c-a)];
        }
        //Формируем элементы от максимального элемента второго слагаемого до
        //максимального элемента первого слагаемого как элементы первого слагаемого
        for (i=d;i<b;i++)
            sum[i]=s1[i];
        //Если максимальное значение первого слагаемого меньше, чем второго
        else
        {
            //Формируем элементы от минимального элемента первого слагаемого до
            //минимального элемента второго слагаемого как элементы первого слагаемого
            for (i=0;i<c-a;i++)
                sum[i]=s1[i];
            //Формируем элементы от минимального элемента второго слагаемого до
            //максимального элемента первого слагаемого как сумму элементов слагаемых
            for (i=c-a;i<d;i++)

```

```

        sum[i]=s1[i]+s2[i-(c-a)];
    //Формируем элементы от максимального элемента первого слагаемого до
    //максимального элемента второго слагаемого как элементы второго слагаемого
    for (i=d;i<b;i++)
        sum[i]=s2[i-(c-a)];
    }
}
//Если минимальное значение первого слагаемого больше, чем второго
else
{
    //Если максимальное значение первого слагаемого больше, чем второго
    if (b>d)
    {
        //Формируем элементы от минимального элемента второго слагаемого до
        //минимального элемента первого слагаемого как элементы второго слагаемого
        for (i=0;i<a-c;i++)
            sum[i]=s2[i];
        //Формируем элементы от минимального элемента первого слагаемого до
        //максимального элемента второго слагаемого как сумму элементов слагаемых
        for (i=a-c;i<d-a+1;i++)
            sum[i]=s1[i-(a-c)]+s2[i];
    }
    //Формируем элементы от максимального элемента первого слагаемого до
    //максимального элемента первого слагаемого как элементы первого слагаемого
    for (i=d-a+1;i<bf;i++)
        sum[i]=s1[i-(a-c)];
    }
    //Если максимальное значение первого слагаемого меньше, чем второго
    else
    {
        //Формируем элементы от минимального элемента второго слагаемого до
        //минимального элемента первого слагаемого как элементы второго слагаемого
        for (i=0;i<a-c;i++)
            sum[i]=s2[i];
        //Формируем элементы от минимального элемента первого слагаемого до
        //максимального элемента первого слагаемого как сумму элементов слагаемых
        for (i=a-c;i<b-a+1;i++)
            sum[i]=s1[i-(a-c)]+s2[i];
    }
    //Формируем элементы от максимального элемента первого слагаемого до
    //максимального элемента второго слагаемого как элементы второго слагаемого
    for (i=b-a+1;i<d;i++)
        sum[i]=s2[i];
    }
}
return sum; //возвращаем указатель на массив результатов
}

//Функция вычисления математического ожидания
float stat_no(int min, //минимум
              int max, //максимум
              float vector[]) //вектор значений для вычисления
{
    float mo=0; //МО
    for (int i=0;i<max-min;i++) //для всех чисел вектора
        mo+=vector[i]*(i+min); //вычисляем МО по формуле
    return mo; //возвращаем результат
}

//Функция вычисления моментов заданного порядка
float stat_moment(int num, //порядок вычисляемого момента
                  int min, //минимум
                  int max, //максимум
                  float vector[]) //вектор значений для вычисления
{

```

```

float mo=0; //вычисляем момент
for (int i=0;i<max-min;i++)//для всех чисел вектора
    mo+=pow(((i+min)-mo),num)*vector[i];//вычисляем требуемый момент по
//ш-ле
return mo; //возвращаем результат
}

//функция передачи в процесс 0 значений массива, вычисленных в остальных
//процессах
void send_rcv(ls *cur,int k,int ProcNum, int CurID)
{
    int bufsize=0; //размер буфера
    float *buf; //указатель на буфер
    int num=0; //счетчик
    //Для всех процессов кроме 0
    if (CurID!=0)
    {
        //Отправляем число передаваемых элементов
        MPI_Send(&k,1,MPI_INT,0,1,MPI_COMM_WORLD);
        //Передаем элементы массива
        MPI_Send(cur->array,k,MPI_FLOAT,0,1,MPI_COMM_WORLD);
        //Удаляем массив
        delete (cur->array);
    }
    //Для процесса 0
    else
    {
        bufsize=0; //обнуляем размер буфера
        for (int z=1;z<ProcNum;z++) //для всех процессов
        {
            //Принимаем размер получаемого массива
            MPI_Recv(&bufsize,1,MPI_INT,z,1,MPI_COMM_WORLD,NULL);
            //Выделяем буфер приема
            buf=new float[bufsize];
            //Получаем массив
            MPI_Recv(buf,bufsize,MPI_FLOAT,z,1,MPI_COMM_WORLD,NULL);
            num=0; //обнуляем счетчик
            for (int j=0;j<bufsize;j++) //для всех полученных элементов
            {
                //записываем их на нужные места в массиве 0-го процесса
                cur->array[num+z]=buf[j];
                num+=ProcNum;
            }
            delete (buf); //очищаем буфер
        }
    }
}

//функция рассылки ЛС из процесса #0 всем остальным процессам
//для ЛС с массивами целых чисел
void synchronize(int iteration,//Число элементов ЛС
                 int arg1[],//Интервал рассылаемых значений
                 ls * begin,//Указатель на начало ЛС
                 int CurID)//Идентификатор процессса
{
    int cursize=0;//Переход в начало ЛС
    //Отправка из процесса 0 f[k] массива
    if (CurID==0)
    {
        cur=begin;
        for (int k=0;k<iteration;k++)//для всех элементов ЛС
        {
            //вычисление текущего размера массива

```

```

cursize=arg1[k];
//Рассылка размера всем процессам
MPI_Bcast(&cursize,1,MPI_INT,0,MPI_COMM_WORLD);
//Рассылка всех элементов массива
MPI_Bcast(cur->array,cursize,MPI_FLOAT,0,MPI_COMM_WORLD);
//Переход к следующему элементу ЛС
cur=cur->next;
}

}

//Прием всеми процессами кроме 0 f[k] массивов
else
{
    cur=begin;//Переход в начало ЛС
    for (int k=0;k<iteration;k++)//для всех элементов ЛС
    {
        //Прием размера принимаемого массива
        MPI_Bcast(&cursize,1,MPI_INT,0,MPI_COMM_WORLD);
        //Выделение памяти под принимаемый массив
        cur->array=new float[cursize];
        //Прием всего массива
        MPI_Bcast(cur->array,cursize,MPI_FLOAT,0,MPI_COMM_WORLD);
        //Переход к следующему элементу
        cur=cur->next;
    }
}

//Функция рассылки ЛС из процесса #0 всем остальным процессам
//Функция перегружена в ЛС для использования с массивами интервалов
void synchronize(
{
    int iteration,//Число элементов ЛС
    m_interval arg1[],//Интервал рассылаемых значений
    ls * begin, //Указатель на начало ЛС
    int CurID) //Идентификатор процесса

{
    //Отправка из процесса 0 f[k] массива
    int cursize=0;
    if (CurID==0)
    {
        cur=begin; //Переход в начало ЛС
        for (int k=0;k<iteration;k++) //для всех элементов ЛС
        {
            //вычисление текущего размера массива
            cursize=arg1[k].max-arg1[k].min+1;
            //Рассылка размера всем процессам
            MPI_Bcast(&cursize,1,MPI_INT,0,MPI_COMM_WORLD);
            //Рассылка всех элементов массива
            MPI_Bcast(cur->array,cursize,MPI_FLOAT,0,MPI_COMM_WORLD);
            //Переход к следующему элементу ЛС
            cur=cur->next;
        }
    }
    //Прием всеми процессами кроме 0 f[k] массивов
    else
    {
        cur=begin; //Переход в начало ЛС
        for (int k=0;k<iteration;k++) //для всех элементов ЛС
        {
            //Прием размера принимаемого массива
            MPI_Bcast(&cursize,1,MPI_INT,0,MPI_COMM_WORLD);
            //Выделение памяти под принимаемый массив
            cur->array=new float[cursize];
            //Прием всего массива
            MPI_Bcast(cur->array,cursize,MPI_FLOAT,0,MPI_COMM_WORLD);
        }
    }
}

```

```

    //Переход к следующему элементу
    cur=cur->next;
}
}

```

## Заголовочный модуль:

```

#include <math.h>

#define sel_num 2

struct m_interval
{
    int min;
    int max;
};

struct ls
{
    ls *next;
    float *array;
};

ls* add(ls**);
float run(int ix1,int ix2, ls* start);

float calculate_f(int,int,int);
void calculate_k_tilda(int [],m_interval k_tilda[]);

void calculate_k_duga(int ,int ix1,int ix2,m_interval res[],m_interval
k_tilda[]);
void calculate_k_dugaor(m_interval res[],m_interval k_tilda[]);

float calculate_f1(int k_tilda,struct m_interval &,int [], float
p_array[],int,m_interval[]);
float calculate_f2(int k_tilda,struct m_interval &,int [], float
p_array[],int,m_interval[]);
float calculate_f3(int k_tilda,struct m_interval &,int [], float
p_array[],int,m_interval[]);
float calculate_f4(int k_tilda,struct m_interval &,int [], float
p_array[],int,m_interval[]);
float calculate_f5(int k_tilda,struct m_interval &,int [], float
p_array[],int,m_interval[]);
float calculate_f6(int k_tilda,struct m_interval &,int [], float
p_array[],int,m_interval[]);
float calculate_f7(int k_tilda,struct m_interval &,int [], float
p_array[],int,m_interval[]);
float calculate_f8(int k_tilda,struct m_interval &,int [], float
p_array[],int,m_interval[]);
float calculate_f9(int k_tilda,struct m_interval &,int [], float
p_array[],int,m_interval[]);
float calculate_f10(int k_tilda,struct m_interval &,int [], float
p_array[],int,m_interval[]);
float calculate_f11(int k_tilda,struct m_interval &,int [], float
p_array[],int);
float calculate_f11_q(int k_tilda,struct m_interval &,int [], float
p_array[],int);

float calculate_fi(int number,int k_tilda,struct m_interval &k_tilda_int,int
k_array[], float p_array[],int FlagOrder,m_interval[]);

```

```

float calculate_qi(int num,int,int ix1,int ix2,int ix3,m_interval
k_tilda[],m_interval k_duga[]);
float calculate_qi_or(int num,int,int ix1,int ix2,int ix3,m_interval
k_tilda[],m_interval k_duga[]);
float * calculate_mult(int ix1, int ix2,m_interval k_duga[],int kmax[]);
float * calculate_sum (float *s1, float *s2,int a,int b,int c,int d);
float stat_mo(int min,int max,float vector[]);
float stat_moment(int num, int min,int max, float vector[]);
void snd_rcv(ls *cur,int k,int ProcNum, int CurID);
void synchronize(int iteration,int arg1[], ls * cur, int CurID);
void synchronize(int,m_interval arg1[], ls *, int);

```

Далее представлен текст главного модуля визуального интерфейса и тексты модулей для отображения получаемых зависимостей.

### Главный модуль:

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Unit2, Unit3, Unit4, Unit5, Unit6, ComCtrls, StrUtils, Grids,
  StdCtrls, ValEdit, Buttons, ExtCtrls, Spin;

type
  TForm1 = class(TForm)
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    Button1: TButton;
    Button3: TButton;
    ValueListEditor1: TValueListEditor;
    ValueListEditor2: TValueListEditor;
    Button2: TButton;
    Button4: TButton;
    Button5: TButton;
    Edit1: TEdit;
    Button6: TButton;
    SpinEdit1: TSpinEdit;
    ListView1: TListView;
    BitBtn1: TBitBtn;
    Button8: TButton;
    RadioGroup1: TRadioGroup;
    CheckBox1: TCheckBox;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure Button6Click(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button8Click(Sender: TObject);
  private

```

```

{ Private declarations }
public
{ Public declarations }
end;

var
Form1: TForm1;
kmax:array [0..27] of integer;
prob:array [0..9] of extended;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
var
i:integer;
begin
end;

procedure TForm1.Button1Click(Sender: TObject);
var
filename :string;
filehnd:TextFile;
buf:string;
i:integer;
begin
if (OpenDialog1.Execute) then
begin
filename:=OpenDialog1.FileName;
AssignFile(filehnd,filename);
Reset(filehnd);
for i:=0 to 27 do begin
Read(filehnd,kmax[i]);
ValueListEditor1.Cells[1,i+1]:=inttostr(kmax[i]);
end;
CloseFile(filehnd);
end;
end;

procedure TForm1.Button5Click(Sender: TObject);
var
ListIItem: TListItem;
begin
ListIItem:=ListView1.Items.Add;
ListIItem.Caption:=Edit1.Text;
ListIItem.SubItems.Add(SpinEdit1.Text);
end;

procedure TForm1.Button6Click(Sender: TObject);
begin
if ListView1.Selected<>nil then
ListView1.Selected.Delete;
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
var
DataFile:textfile;
i:integer;
MyCanvas:TCanvas;

```

```

temp:String;
P: PChar;

begin
//Create cfg file
  Button8.Enabled:=false;
  //TFileStream.Create( 'mpi.cfg', (fmCreate or fmOpenReadWrite) );
  AssignFile(DataFile, 'mpi.cfg');
  Rewrite(DataFile);
  WriteLn(DataFile,'exe parallel.exe');
  Write(DataFile,'args');
  for i:=0 to 27 do Write(DataFile,' ',ValueListEditor1.Cells[1,i+1]);
  for i:=0 to 9 do
  begin
    Write(DataFile,' ',AnsiReplaceStr(ValueListEditor2.Cells[1,i+1],',','.'));
  end;
  Writeln(DataFile,' ',RadioGroup1.ItemIndex);
  Writeln(DataFile,'hosts');
  for i:=0 to ListView1.Items.Count-1 do
  Writeln(DataFile,ListView1.Items.Item[i].Caption,
  ',ListView1.Items.Item[i].SubItems[0]);
  CloseFile(DataFile);
//Execute mpi
  if (CheckBox1.Checked) then WinExec(PChar('mpirun -localonly
mpi.cfg'),SW_HIDE)
  else WinExec(PChar('mpirun mpi.cfg'),SW_HIDE);

//Process results
  Button8.Enabled:=true;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
filename :string;
filehnd:TextFile;
buf:string;
i:integer;
begin
if (OpenDialog1.Execute) then
begin

  filename:=OpenDialog1.FileName;
  AssignFile(filehnd,filename);
  Reset(filehnd);
  for i:=0 to 9 do begin
    Read(filehnd,buf[i]);
    ValueListEditor2.Cells[1,i+1]:=floattostr(buf[i]);
  end;

  CloseFile(filehnd);

end;
end;

procedure TForm1.Button8Click(Sender: TObject);
begin
  Form2:=TForm2.Create(nil);
  Form3:=TForm3.Create(nil);
  Form4:=TForm4.Create(nil);
  Form5:=TForm5.Create(nil);
  Form6:=TForm6.Create(nil);
  Form2.Show;
  Form3.Show;

```

```

Form4.Show;
Form5.Show;
Form6.Show;

end;

end.

Модули для отображения:
1)

unit Unit2;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm2 = class(TForm)
    Label2: TLabel;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form2: TForm2;

implementation

uses Unit4;

{$R *.dfm}

procedure TForm2.Button1Click(Sender: TObject);
var
  filename :string;
  filehnd:TextFile;
  i:integer;
  temp,h_max,w_max,h_scale,w_scale:extended;
  a,b,d:array of integer;
  c:array of extended;
  num:integer;
  no,disp,ex,assym:extended;
begin
  AssignFile(filehnd,'1.res');
  Reset(filehnd);

  Read(filehnd,num);

  SetLength(a, num);
  SetLength(b, num);
  SetLength(c, num);
  SetLength(d, num);

  for i:=0 to num-1 do
    Read(filehnd,d[i]);

```

```

for i:=0 to num-1 do
  Read(filehnd,c[i]);

//Read(filehnd,mo,disp,ex,assym);

h_max:=0;
w_max:=0;

for i:=0 to num-1 do
begin
  if c[i]>h_max then h_max:=c[i];
  if d[i]>w_max then w_max:=d[i];
end;

h_scale:=(Form4.Height-200)/h_max;
w_scale:=(Form4.Width-100)/w_max;

for i:=0 to num-1 do
begin
  b[i]:=Round(c[i]*h_scale)+150;
  a[i]:=Round(d[i]*w_scale);
end;

CloseFile(filehnd);

Canvas.Brush.Color:=clWhite;
Canvas.Rectangle(50,30,Form4.Width-50,Form4.Height-120);

Canvas.Brush.Color := clRed;
Canvas.Pen.Color:= clRed;
Canvas.Brush.Style := bsSolid;

//Points

for i:=0 to num-1 do
begin
  Canvas.Ellipse(a[i]+30,Form4.Height-b[i],a[i]+30+8,Form4.Height-b[i]-8);
  Canvas.MoveTo(a[i]+4+30,Form4.Height-b[i]-4);
  if (i<>num-1) then Canvas.LineTo(a[i+1]+4+30,Form4.Height-b[i+1]-4);
end;

Canvas.Brush.Color := clBtnFace;
Canvas.Pen.Color:= clGray;

//Vert_lines

for i:=0 to num-1 do
begin
  Canvas.MoveTo(a[i]+4+30,30);
  if i=0 then
  begin
    Canvas.LineTo(a[i]+30+4,Form4.Height-120);
    Canvas.TextOut(a[i]+30,Form4.Height-110,inttostr(d[i]));
  end;
  if (i>0) and ((abs(a[i]-a[i-1]))>10) then
  begin
    if ((i mod 2)=0) then
    begin
      Canvas.LineTo(a[i]+30+4,Form4.Height-120);
      Canvas.TextOut(a[i]+30,Form4.Height-110,inttostr(d[i]));
    end;
  end;
end;

```

```

//Horiz_lines

for i:=0 to num-1 do
begin
  Canvas.MoveTo(50,Form4.Height-b[i]-4);
  if (i=0) then
  begin
    Canvas.LineTo(Form4.Width-50,Form4.Height-b[i]-4);
    Canvas.TextOut(0,Form4.Height-b[i]-4,floattostr(c[i]));
  end;
  if (i>0) and ((abs(b[i]-b[i-1]))>5) then
  begin
    if ((i mod 2)=0) then
    begin
      Canvas.LineTo(Form4.Width-50,Form4.Height-b[i]-4);
      Canvas.TextOut(0,Form4.Height-b[i]-4,floattostr(c[i]));
    end;
  end;
end;

Label2.Caption:='MO='+FloatToStr(mo); Covariation='+FloatToStr(disp)+';
Assym='+FloatToStr(assym); // Ex='+FloatToStr(ex);
end;
end.
```

2)

```

unit Unit3;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm3 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    procedure FormPaint(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form3: TForm3;

implementation

uses Unit2, Unit4;

{$R *.dfm}

procedure TForm3.FormPaint(Sender: TObject);
var
  filename :string;
  filehnd:TextFile;
  i:integer;
  temp,h_max,w_max,h_scale,w_scale:extended;
  a,b,d:array of integer;
```

```

c:array of extended;
num:integer;
mo,disp,ex,assym:extended;
begin
  AssignFile(filehnd,'2.res');
  Reset(filehnd);

  Read(filehnd,num);

  SetLength(a, num);
  SetLength(b, num);
  SetLength(c, num);
  SetLength(d, num);

  for i:=0 to num-1 do
    Read(filehnd,d[i]);

  for i:=0 to num-1 do
    Read(filehnd,c[i]);

  //Read(filehnd,mo,disp,ex,assym);

  h_max:=0;
  w_max:=0;

  for i:=0 to num-1 do
  begin
    if c[i]>h_max then h_max:=c[i];
    if d[i]>w_max then w_max:=d[i];
  end;

  h_scale:=(Form4.Height-200)/h_max;
  w_scale:=(Form4.Width-100)/w_max;

  for i:=0 to num-1 do
  begin
    b[i]:=Round(c[i]*h_scale)+150;
    a[i]:=Round(d[i]*w_scale);
  end;

  CloseFile(filehnd);

  Canvas.Brush.Color:=clWhite;
  Canvas.Rectangle(50,30,Form4.Width-50,Form4.Height-120);

  Canvas.Brush.Color := clRed;
  Canvas.Pen.Color:= clRed;
  Canvas.Brush.Style := bsSolid;

  //Points

  for i:=0 to num-1 do
  begin
    Canvas.Ellipse(a[i]+30,Form4.Height-b[i],a[i]+30+8,Form4.Height-b[i]-8);
    Canvas.MoveTo(a[i]+4+30,Form4.Height-b[i]-4);
    if (i<>num-1) then Canvas.LineTo(a[i+1]+4+30,Form4.Height-b[i+1]-4);
  end;

  Canvas.Brush.Color := clBtnFace;
  Canvas.Pen.Color:= clGray;

  //Vert_lines

  for i:=0 to num-1 do

```

```

begin
  Canvas.MoveTo(a[i]+4+30,30);
  if i=0 then
  begin
    Canvas.LineTo(a[i]+30+4,Form4.Height-120);
    Canvas.TextOut(a[i]+30,Form4.Height-110,inttostr(d[i]));
  end;
  if (i>0) and ((abs(a[i]-a[i-1]))>10)then
  begin
    if ((i mod 2)=0) then
    begin
      Canvas.LineTo(a[i]+30+4,Form4.Height-120);
      Canvas.TextOut(a[i]+30,Form4.Height-110,inttostr(d[i]));
    end;
  end;
end;

//Horiz_lines

for i:=0 to num-1 do
begin
  Canvas.MoveTo(50,Form4.Height-b[i]-4);
  if (i=0) then
  begin
    Canvas.LineTo(Form4.Width-50,Form4.Height-b[i]-4);
    Canvas.TextOut(0,Form4.Height-b[i]-4,floattosstr(c[i]));
  end;
  if (i>0) and ((abs(b[i]-b[i-1]))>5)then
  begin
    if ((i mod 2)=0) then
    begin
      Canvas.LineTo(Form4.Width-50,Form4.Height-b[i]-4);
      Canvas.TextOut(0,Form4.Height-b[i]-4,floattosstr(c[i]));
    end;
  end;
end;

Label2.Caption:='MO='+FloatToStr(mo)+'; Covariation='+FloatToStr(disp)+';
Assym='+FloatToStr(assym); // Ex='+FloatToStr(ex);
end;
end.

```

3)

```

unit Unit4;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm4 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    procedure FormPaint(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

```

```

var
  TForm4: TForm4;

implementation

uses Unit2;

{$R *.dfm}

procedure TForm4.FormPaint(Sender: TObject);
var
  filename :string;
  filehnd:TextFile;
  i:integer;
  temp,h_max,w_max,h_scale,w_scale:extended;
  a,b,d:array of integer;
  c:array of extended;
  num:integer;
  mo,disp,ex,assym:extended;
begin
  AssignFile(filehnd,'3.res');
  Reset(filehnd);

  Read(filehnd,num);

  SetLength(a, num) ;
  SetLength(b, num) ;
  SetLength(c, num) ;
  SetLength(d, num) ;

  for i:=0 to num-1 do
    Read(filehnd,d[i]);

  for i:=0 to num-1 do
    Read(filehnd,c[i]);

  //Read(filehnd,mo,disp,ex,assym);

  h_max:=0;
  w_max:=0;

  for i:=0 to num-1 do
  begin
    if c[i]>h_max then h_max:=c[i];
    if d[i]>w_max then w_max:=d[i];
  end;

  h_scale:=(Form4.Height-200)/h_max;
  w_scale:=(Form4.Width-100)/w_max;

  for i:=0 to num-1 do
  begin
    b[i]:=Round(c[i]*h_scale)+150;
    a[i]:=Round(d[i]*w_scale);
  end;

  CloseFile(filehnd);

  Canvas.Brush.Color:=clWhite;
  Canvas.Rectangle(50,30,Form4.Width-50,Form4.Height-120);

  Canvas.Brush.Color := clRed;
  Canvas.Pen.Color:= clRed;
  Canvas.Brush.Style := bsSolid;

```

```

//Points

for i:=0 to num-1 do
begin
  Canvas.Ellipse(a[i]+30,Form4.Height-b[i],a[i]+30+8,Form4.Height-b[i]-8);
  Canvas.MoveTo(a[i]+4+30,Form4.Height-b[i]-4);
  if (i<>num-1) then Canvas.LineTo(a[i+1]+4+30,Form4.Height-b[i+1]-4);
end;

Canvas.Brush.Color := clBtnFace;
Canvas.Pen.Color:= clGray;

//Vert_lines

for i:=0 to num-1 do
begin
  Canvas.MoveTo(a[i]+4+30,30);
  if i=0 then
  begin
    Canvas.LineTo(a[i]+30+4,Form4.Height-120);
    Canvas.TextOut(a[i]+30,Form4.Height-110,inttostr(d[i]));
  end;
  if (i>0) and ((abs(a[i]-a[i-1]))>10) then
  begin
    if ((i mod 2)=0) then
    begin
      Canvas.LineTo(a[i]+30+4,Form4.Height-120);
      Canvas.TextOut(a[i]+30,Form4.Height-110,inttostr(d[i]));
    end;
  end;
end;

//Horiz_lines

for i:=0 to num-1 do
begin
  Canvas.MoveTo(50,Form4.Height-b[i]-4);
  if (i=0) then
  begin
    Canvas.LineTo(Form4.Width-50,Form4.Height-b[i]-4);
    Canvas.TextOut(0,Form4.Height-b[i]-4,floattostr(c[i]));
  end;
  if (i>0) and ((abs(b[i]-b[i-1]))>5) then
  begin
    if ((i mod 2)=0) then
    begin
      Canvas.LineTo(Form4.Width-50,Form4.Height-b[i]-4);
      Canvas.TextOut(0,Form4.Height-b[i]-4,floattostr(c[i]));
    end;
  end;
end;

Label2.Caption:='MO='+FloatToStr(mo)+'; Covariation='+FloatToStr(disp)+';
Assym='+FloatToStr(assym); // Ex='+FloatToStr(ex);
end;

end.

```

4)

```
unit Unit5;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm5 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    procedure FormPaint(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form5: TForm5;

implementation

uses Unit2, Unit4;

{$R *.dfm}

procedure TForm5.FormPaint(Sender: TObject);
var
  filename :string;
  filehnd:TextFile;
  i:integer;
  temp,h_max,w_max,h_scale,w_scale:extended;
  a,b,d:array of integer;
  c:array of extended;
  num:integer;
  mo,disp,ex,assym:extended;
begin
  AssignFile(filehnd,'4.res');
  Reset(filehnd);

  Read(filehnd,num);

  SetLength(a, num);
  SetLength(b, num);
  SetLength(c, num);
  SetLength(d, num);

  for i:=0 to num-1 do
    Read(filehnd,d[i]);

  for i:=0 to num-1 do
    Read(filehnd,c[i]);

  //Read(filehnd,mo,disp,ex,assym);

  h_max:=0;
  w_max:=0;

  for i:=0 to num-1 do
```

```

begin
  if c[i]>h_max then h_max:=c[i];
  if d[i]>w_max then w_max:=d[i];
end;

h_scale:=(Form4.Height-200)/h_max;
w_scale:=(Form4.Width-100)/w_max;

for i:=0 to num-1 do
begin
  b[i]:=Round(c[i]*h_scale)+150;
  a[i]:=Round(d[i]*w_scale);
end;

CloseFile(filehnd);

Canvas.Brush.Color:=clWhite;
Canvas.Rectangle(50+30,Form4.Width-50,Form4.Height-120);

Canvas.Brush.Color := clRed;
Canvas.Pen.Color:= clRed;
Canvas.Brush.Style := bsSolid;

//Points

for i:=0 to num-1 do
begin
  Canvas.Ellipse(a[i]+30,Form4.Height-b[i],a[i]+30+8,Form4.Height-b[i]-8);
  Canvas.MoveTo(a[i]+4+30,Form4.Height-b[i]-4);
  if (i<>num-1) then Canvas.LineTo(a[i+1]+4+30,Form4.Height-b[i+1]-4);
end;

Canvas.Brush.Color := clBtnFace;
Canvas.Pen.Color:= clGray;

//Vert_lines

for i:=0 to num-1 do
begin
  Canvas.MoveTo(a[i]+4+30,30);
  if i=0 then
  begin
    Canvas.LineTo(a[i]+30+4,Form4.Height-120);
    Canvas.TextOut(a[i]+30,Form4.Height-110,inttostr(d[i]));
  end;
  if (i>0) and ((abs(a[i]-a[i-1]))>10) then
  begin
    if ((i mod 2)=0) then
    begin
      Canvas.LineTo(a[i]+30+4,Form4.Height-120);
      Canvas.TextOut(a[i]+30,Form4.Height-110,inttostr(d[i]));
    end;
  end;
end;

//Horiz_lines

for i:=0 to num-1 do
begin
  Canvas.MoveTo(50,Form4.Height-b[i]-4);
  if (i=0) then
  begin
    Canvas.LineTo(Form4.Width-50,Form4.Height-b[i]-4);
    Canvas.TextOut(0,Form4.Height-b[i]-4,floattostr(c[i]));
  end;
end;

```

```

    end;
    if (i>0) and ((abs(b[i]-b[i-1]))>5) then
    begin
      if ((i mod 2)=0) then
      begin
        Canvas.LineTo(Form4.Width-50,Form4.Height-b[i]-4);
        Canvas.TextOut(0,Form4.Height-b[i]-4,floattosstr(c[i]));
      end;
    end;
  end;

Label2.Caption:='MO='+FloatToStr(mo)+'; Covariation='+FloatToStr(disp)+';
Assym='+FloatToStr(assym); // Ex='+FloatToStr(ex);
end;
end.

5)

unit Unit6;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm6 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    procedure FormPaint(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form6: TForm6;

implementation

uses Unit2, Unit4;

{$R *.dfm}

procedure TForm6.FormPaint(Sender: TObject);
var
  filename :string;
  filehnd:TextFile;
  i:integer;
  temp,h_max,w_max,h_scale,w_scale:extended;
  a,b,d:array of integer;
  c:array of extended;
  num:integer;
  mo,disp,ex,assym:extended;
begin
  AssignFile(filehnd,'5.res');
  Reset(filehnd);
  Read(filehnd,num);

```

```

SetLength(a, num);
SetLength(b, num);
SetLength(c, num);
SetLength(d, num);

for i:=0 to num-1 do
  Read(filehnd,d[i]);

for i:=0 to num-1 do
  Read(filehnd,c[i]);

//Read(filehnd,mc,disp,ex,assym);

h_max:=0;
w_max:=0;

for i:=0 to num-1 do
begin
  if c[i]>h_max then h_max:=c[i];
  if d[i]>w_max then w_max:=d[i];
end;

h_scale:=(Form4.Height-200)/h_max;
w_scale:=(Form4.Width-100)/w_max;

for i:=0 to num-1 do
begin
  b[i]:=Round(c[i]*h_scale)+150;
  a[i]:=Round(d[i]*w_scale);
end;

CloseFile(filehnd);

Canvas.Brush.Color:=clWhite;
Canvas.Rectangle(50,30,Form4.Width-50,Form4.Height-120);

Canvas.Brush.Color := clRed;
Canvas.Pen.Color:= clRed;
Canvas.Brush.Style := bsSolid;

//Points

for i:=0 to num-1 do
begin
  Canvas.Ellipse(a[i]+30,Form4.Height-b[i],a[i]+30+8,Form4.Height-b[i]-8);
  Canvas.MoveTo(a[i]+4+30,Form4.Height-b[i]-4);
  if (i>num-1) then Canvas.LineTo(a[i+1]+4+30,Form4.Height-b[i+1]-4);
end;

Canvas.Brush.Color := clBtnFace;
Canvas.Pen.Color:= clGray;

//Vert_lines

for i:=0 to num-1 do
begin
  Canvas.MoveTo(a[i]+4+30,30);
  if i=0 then
  begin
    Canvas.LineTo(a[i]+30+4,Form4.Height-120);
    Canvas.TextOut(a[i]+30,Form4.Height-110,inttostr(d[i]));
  end;
  if (i>0) and ((abs(a[i]-a[i-1]))>10) then
  begin

```

```

if ((i mod 2)=0) then
begin
  Canvas.LineTo(a[i]+30+4,Form4.Height-120);
  Canvas.TextOut(a[i]+30,Form4.Height-110,inttostr(d[i]));
end;
end;
end;

//Horiz_lines

for i:=0 to num-1 do
begin
  Canvas.MoveTo(50,Form4.Height-b[i]-4);
  if (i=0) then
begin
  Canvas.LineTo(Form4.Width-50,Form4.Height-b[i]-4);
  Canvas.TextOut(0,Form4.Height-b[i]-4,floattostr(c[i]));
end;
if (i>0) and ((abs(b[i]-b[i-1]))>5)then
begin
  if ((i mod 2)=0) then
  begin
    Canvas.LineTo(Form4.Width-50,Form4.Height-b[i]-4);
    Canvas.TextOut(0,Form4.Height-b[i]-4,floattostr(c[i]));
  end;
end;
end;
end;

Label2.Caption:='MO='+FloatToStr(mo)+'; Covariation='+FloatToStr(disp)+';
Assym='+FloatToStr(assym);// Ex='+FloatToStr(ex);
end;
end.

```



