Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра Вычислительных систем

Допустить к защите

Зав.каф. _____Курносов М. Г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Методы организации отказоустойчивого выполнения параллельных программ

Магистерская диссертация

по направлению 09.04.01 «Информатика и вычислительная техника»

Студент _____ / Гайдай А. В. /

Руководитель _____ / Курносов М. Г. /

Новосибирск 2018 г.

Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

КАФЕДРА

вычислительных систем

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
МАГИСТРАНТА

СТУДЕНТА Гайдая А. В. ГРУППЫ МГ-165

УТВЕРЖДАЮ

« ___ » _____ 20__ г.

Зав. кафедрой

_____ / *Курносов М. Г.* /

Новосибирск 2018 г.

1. Тема выпускной квалификационной работы магистранта
Методы организации отказоустойчивого выполнения параллельных программ

утверждена приказом СибГУТИ от «31» октября 2016 г. № 10/125А-16

2.Срок сдачи студентом законченной работы «20» июня 2018 г.

3.Исходные данные к работе
1 Хорошевский, В.Г. Архитектура вычислительных систем [Текст] / В.Г. Хорошевский. – М.: МГТУ им. Н. Э. Баумана, 2008. – 520 с.

2 Хорошевский, В.Г. Распределенные вычислительные системы с программируемой структурой [Текст] / В.Г. Хорошевский // Вестник СибГУТИ. – 2010. – № 2. – С. 3-41.

4.Содержание пояснительной записки (перечень подлежащих разработке вопросов)	**Сроки выполнения по разделам**
Введение	5.02 – 17.02
Вычислительные системы	18.02 – 20.04
Подходы к организации отказоустойчивости	21.04 – 21.05
Результаты экспериментов	22.05 – 10.06
Заключение	11.06 – 20.06

Дата выдачи задания «5» февраля 2018 г.
Руководитель _____
подпись

Задание принял к исполнению «5» февраля 2018 г.
Студент _____
подпись

АННОТАЦИЯ

Выпускная квалификационная работа Гайдая А. В. по теме «Методы организации отказоустойчивого выполнения параллельных программ»

Объём работы *57* страниц, на которых размещены *14* рисунков и *2* таблицы. При написании работы использовалось *34* источника.

Ключевые слова: *MPI*, параллелизм, вычислительные системы, большемасштабность, отказоустойчивость, живучесть, метод Гаусса.

Работа выполнена на кафедре вычислительных систем СибГУТИ.

Ожидаемые результаты: разработанная *MPI*-программа, реализующая отказоустойчивую параллельную версию алгоритма решения системы линейных алгебраических уравнений методом Гаусса.

Graduation thesis abstract

of A.V. Gaidai on the theme «Methods of fault-tolerant program execution organization»

The paper consists of *57* pages, with *14* figures and *2* tables/charts/diagrams. While writing the thesis *34* reference sources were used.

Keywords: *MPI*, parallelism, computing systems, large scale, fault tolerance, survivability, Gaussian elimination.

The thesis was written at the department of Computer Systems of SibSUTIS.

Scientific supervisor is Associate Professor, Doctor of Science, Kurnosov Mikhail.

The goal/subject of the paper is development of a fault-tolerant version of the algorithm of solving a system of linear algebraic equations using the Gaussian elimination.

Tasks: development of a fault-tolerant version of the algorithm of solving a system of linear algebraic equations using the Gaussian elimination; analysis of the developed implementation; assessment of overhead costs for organization of fault tolerance.

Results: implementation of a fault-tolerant version of the algorithm, of solving a system of linear algebraic equations using the Gaussian elimination; fault tolerance is fully achieved.

Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

ОТЗЫВ

на выпускную квалификационную работу Гайдая А. В.
по теме «Методы организации отказоустойчивого выполнения параллельных программ»

В выпускной квалификационной работе перед Гайдаем А.В. стояла задача разработать отказоустойчивую реализацию алгоритма решения системы линейных алгебраических уравнений методом Гаусса. С данной задачей А.В. Гайдай справился частично. Выполнена реализация метода Гаусса в стандарте MPI и начальная версия с использованием ULFM MPI. Эксперименты с отказоустойчивой версией программы проведены в ограниченном объеме.

Считаю, что выпускная квалификационная работа магистра Гайдая А.В. заслуживает оценки «хорошо», а Гайдай А.В. присвоения квалификации магистр по направлению 09.04.01 «Информатика и вычислительная техника».

Компетенции		Уровень сформированности компетенций		
		высокий	средний	низкий
Общекультурные	ОК-1 Способность совершенствовать и развивать свой интеллектуальный и общекультурный уровень			
	ОК-3 Способность к самостоятельному обучению новым методам исследования, к изменению научного и научно-производственного профиля своей профессиональной деятельности			
	ОК-4 Способность заниматься научными исследованиями			
	ОК-5 Использование на практике умений и навыков в организации			

	исследовательских и проектных работ, в управлении коллективом			
	ОК-6 Способность проявлять инициативу, в том числе в ситуациях риска, брать на себя всю полноту ответственности			
	ОК-9 Умение оформлять отчеты о проведенной научно-исследовательской работе и подготавливать публикации по результатам исследования			
Общепрофессиональные	ОПК-3 Способность анализировать и оценивать уровни своих компетенций в сочетании со способностью и готовностью к саморегулированию дальнейшего образования и профессиональной мобильности			
	ОПК-5 Владение методами и средствами получения, хранения, переработки и трансляции информации посредством современных компьютерных технологий, в том числе в глобальных компьютерных сетях			
Профессиональные	ПК-2 знанием методов научных исследований и владение навыками их проведения			
	ПК-7 применением перспективных методов исследования и решения профессиональных задач на основе знания мировых тенденций развития вычислительной техники и информационных технологий			
	ПК-17 способностью к организации промышленного тестирования создаваемого программного обеспечения			

Работа имеет практическую ценность		Тема предложена предприятием	
Работа имеет теоретическую значимость		Тема предложена студентом	
Работа внедрена		Тема является фундаментальной	
Рекомендую работу к опубликованию		Рекомендую студента в аспирантуру	
Работа выполнена в рамках гранта НИОКР		Имеются публикации по теме работы	

Профессор Кафедры ВС, д.т.н. _____ Курносов М.Г.

Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Форма утверждена научно-методическим
советом университета
протокол № 4 от 18.03.2008 г.

РЕЦЕНЗИЯ
на магистерскую диссертацию

Студента Гайдая Анатолия Валерьевича.

По специальности «вычислительные машины, системы, комплексы, сети», группы МГ-165, по направлению 09.04.01. Информатика и вычислительная техника (ИВТ).

Тема магистерской диссертации «Методы организации отказоустойчивого выполнения параллельных программ».

Объём работы 57 страниц, на которых размещены 14 рисунков и 2 таблицы. При написании работы использовалось 34 источника.

Представленная работа направлена на исследование способов организации отказоустойчивости при выполнении *MPI*-программ и на реализацию отказоустойчивой версии параллельного алгоритма с помощью расширения *ULFM*. В качестве демонстрационного примера используется классическая задача линейной алгебры – решение СЛАУ методом Гаусса. Оформление рукописи соответствует требованиям, предъявляемым для данной работы. Восприятие текстового и иллюстрационного материала оценивается положительно. Магистерская диссертация представляет собой логично построенное научное исследование, содержание работы соответствует заданию.

Тема диссертации является весьма актуальной и востребованной. Стандарт *MPI* является основным средством разработки параллельных программ. Расширение *ULFM* является наиболее распространённым и предоставляет

разработчикам средства программной организации отказоустойчивости. Современные высокопроизводительные системы являются большемасштабными и время безотказной работы в них варьируется в пределах нескольких десятков минут до пары часов. Существующие способы организации отказоустойчивости, основанные на использовании контрольных точек (КТ) восстановления становятся малоэффективными, так как время создания согласованной КТ приближается ко времени безотказной работы современных вычислительных систем, на которых выполняются разрабатываемые *MPI*-программы.

Научная ценность работы заключается в исследовании методов организации отказоустойчивости *MPI*-программ и в создании отказоустойчивой версии алгоритма решения СЛАУ методом Гаусса. Параллельная живучая программа решения СЛАУ методом Гаусса реализована в стандарте *MPI* с использованием расширения *ULFM* и экспериментально исследована на вычислительном кластере с сетью связи стандарта *Gigabit Ethernet*.

Магистерская диссертация имеет качественно проработанный материал. Приведен анализ существующих способов организации отказоустойчивости *MPI*-программ. Программно реализована живучая версия алгоритма решения СЛАУ методом Гаусса, показана её эффективность.

Считаю, что данная работа отвечает требованиям, предъявляемым к магистерским диссертациям с оценкой «отлично», а Гайдай А.В. заслуживает присуждения степени магистра по направлению «Информатика и вычислительная техника».

к.т.н., доцент кафедры ПМиК
Нечта Иван Васильевич _____ «__» _____ 2018 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В распределённых системах с многочисленными или сложными компонентами имеется серьёзный риск того, что компонентная ошибка проявится как отказ процесса, который нарушит нормальное выполнение запущенного длительное время приложения [10]. В силу этого и тенденции наращивания производительности вычислительных систем за счёт увеличения количества их вычислителей, становится актуальным задача написания отказоустойчивых параллельных программ по средствам расширения (*ULFM – User Level Failure Mitigation*) для имеющейся реализации стандартного интерфейса передачи сообщений (*OpenMPI*). Расширение *ULFM* предоставляет возможность реконфигурации системы в случае возникновения отказа одного или нескольких процессов.

В данной работе, на примере задачи решения системы линейных алгебраических уравнений (СЛАУ) методом Гаусса, планируется рассмотреть основные подходы организации отказоустойчивости в большемасштабных вычислительных системах (ВС): с использованием контрольных точек на уровне пользователя/операционной системы и без их использования (живучие алгоритмы).

В качестве результатов предполагается:

1. Провести анализ особенностей каждого из рассмотренных подходов.
2. Разработать программную реализацию живучей версии алгоритма решения СЛАУ методом Гаусса, в которой вычислительная часть, в случае возникновения отказа, динамически распределяется на доступное количество рабочих процессов.
3. Построить графики, демонстрирующих безотказную работу алгоритма и его производительность относительно не живучей параллельной версии.

1 Вычислительные системы

Понятие вычислительной системы базируется на модели коллектива вычислителей, на понятиях параллельного алгоритма, программы и архитектуры [33, 18, 30]. Коллектив аппаратурно-программных вычислителей является ВС. Это достаточно общее определение: если посмотреть значение слова «система» в философском словаре, то определение ВС можно конкретизировать. По мнению В. Н. Садовского система (от греч. *systema* – составленное из частей, соединённое) – совокупность элементов, находящихся в отношении и связях между собой и образующих определённую целостность, единство. В середине 20-ого века большое значение для понимания механизмов системы управления приобрели кибернетика и цикл связанных с нею научных и технических дисциплин. Понятие системы органически связано с понятиями целостности, элемента, подсистемы, связи, отношения, структуры и др. Для системы характерно не только наличие связей и отношений между образующими её элементами (определённая организованность), но и неразрывное единство со средой, во взаимоотношении с которой система проявляет свою целостность. Любая система может быть рассмотрена как элемент системы более высокого порядка, в то время как её элементы могут выступать в качестве систем более низкого порядка. Иерархичность, многоуровневость характеризуют строение, морфологию системы и её поведение, функционирование: отдельные уровни системы обуславливают определённые аспекты её поведения, а целостное функционирование оказывается результатом взаимодействия всех её сторон, уровней. Для большинства систем характерно наличие в них процессов передачи информации и управления. К наиболее сложным типам систем относятся *целенаправленные системы*, поведение которых подчинено достижению определённой цели, и *самоорганизующиеся системы*, способные в процессе своего функционирования изменять свою структуру [32]. Таким образом, ВС – это совокупность взаимосвязанных аппаратно-программных элементов (вычислителей), обменивающихся между собой информацией, имеющих

процесс управления, направленный на достижение некоторого результата и образующих определённую целостность, единство.

1.1 Архитектура ВС

Архитектура ВС основывается на имитации коллектива людей-вычислителей. От уровня адекватности такой имитации зависит потенциальные архитектурные возможности ВС.

1.1.1 Модель коллектива вычислителей

Модель коллектива вычислителей сформулирована в [33]. Её основами являются следующие архитектурные принципы:

- параллелизм (*Parallelism, Concurrency*) при обработке информации;
- программируемость (*Programmability, Adaptability*) структуры;
- однородность (*Homogeneity*) вычислителей и структуры.

Под параллелизмом понимается одновременное выполнение элементами ВС собственных задач в процессе обработки информации. Благодаря этому принципу теоретически появляется возможность наращивания производительности ВС за счёт увеличения количества её вычислителей (элементарных машин). Однако, узким местом при таком подходе становится взаимосвязь между узлами ВС (*interconnect*): скорость передачи информации, «объёмность» конструкции.

Стоит заметить, что принцип программируемости структуры ВС является фундаментальным в области средств обработки информации. Под программируемостью структуры подразумевается возможность:

- хранения в коллективе вычислителей описания его изначальной физической структуры;
- предопределённой автоматической (программной) настройки виртуальных конфигураций и их перенастройки в процессе функционирования с целью обеспечения соответствия структурам и параметрам решаемых задач;

- достижения эффективности при заданных условиях эксплуатации.

Уровень развития вычислительной техники позволяет в некоторых областях использовать принцип виртуальной однородности конструкции. Более того, можно использовать неоднородные структуры, ограничившись лишь требованием совместимости вычислителей в коллективе. Однако следует отметить, что при создании высокопроизводительных ВС принцип однородности всё же имеет немалое значение.

1.1.2 Архитектурные свойства ВС

В теории ВС выделяют несколько наиболее важных свойств, которые в той или иной мере могут проявляться в различных реализациях ВС.

Масштабируемость (*Scalability*) ВС. Под масштабируемостью ВС понимается возможность варьирования её производительности за счёт наращивания или сокращения количества входящих в её состав вычислителей. Данное архитектурное свойство позволяет сохранять в течении длительного времени способность ВС адекватно решать сложные задачи. Выполнение этого свойства гарантируется принципами локальности, модульности, распределённости и децентрализованности.

Теоретически свойство наращиваемости производительности предоставляет возможность решать задачи любой заранее заданной сложности. Однако, практическая реализация этой возможности требует, чтобы алгоритм решения задачи удовлетворял условию локальности, а межмодульные обмены информацией слабо влияли на время её решения. Этого можно достичь, применяя методику крупноблочного распараллеливания сложных задач [33].

Универсальность (*Genericity, Generality, Versatility*) ВС. Принято считать, что ЭВМ, основанные на модели вычислителя, являются алгоритмически универсальными, если на них можно без изменения структуры реализовать алгоритм решения любой задачи. ВС – это коллектив вычислителей, каждый из которых обладает свойством универсальности, следовательно, и вся система также является универсальной.

Онтологический принцип холизма гласит: целое всегда есть нечто большее, чем простая сумма его частей. Таким образом, в ВС могут быть реализованы не только любые алгоритмы, доступные ЭВМ, но и параллельные алгоритмы решения сложных задач.

Структурная универсальность ВС вытекает из основных архитектурных принципов коллектива вычислителей. В частности, принцип программируемости структуры предполагает возможность автоматически (программно) создавать специализированные виртуальные конфигурации, которые соответствуют структурам и параметрам решаемых задач.

Структурная универсальность позволяет автоматически подбирать такую конфигурацию из ресурсов ВС, которая максимально бы подходила алгоритму решения определённой задачи. Таким образом, ВС – это средство, в котором диалектически сочетаются противоположные свойства универсальности и специализированности.

Производительность (*Performance, Throughput, Processing power*) ВС. В отличии от ЭВМ, наращивание производительности ВС достигается за счёт не только совершенствования элементной базы, а главным образом благодаря увеличению числа вычислителей, входящих в её состав. Также, стоит отметить, что принцип однородности позволяет осуществлять наращивание ВС простым присоединением дополнительных вычислителей без конструктивных изменений уже имеющихся в системе ЭМ. При этом достигается простота настройка ПО на заданное количество вычислителей в системе, что обеспечивает совместимость ВС различной производительности.

Реконфигурируемость (*Reconfigurability*) ВС. Можно выделить два тип реконфигурации ВС: статическая и динамическая. Под статической реконфигурацией понимается: возможность варьирования числа вычислителей, изменение структуры ВС и её состава; допустимость использования в качестве связей каналов различных типов и т.п. Благодаря статической реконфигурации достигается адаптация системы под область применения на этапе её формирования.

Динамическая реконфигурация ВС достигается за счёт наличия возможности образования в системах виртуальных подсистем, структуры и функциональная часть которых соответствуют входной мультипрограммной ситуации и структурам решаемых задач. Следовательно, динамическая реконфигурация ВС позволяет достигать высокого уровня универсальности, при котором ВС показывает заданную производительность при решении широкого класса задач; реализуются разнообразные режимы функционирования, способы управления вычислительным процессом, структурные схемы и способы обработки информации.

Надёжность и живучесть (*Reliability and Robustness*) ВС. Под надёжностью ВС следует понимать её способность к автоматической настройке и организации функционирования таких структурных схем, которые при отказах и восстановлениях вычислителей сохраняют заданный уровень производительности или, иначе говоря, предоставляют возможность использовать фиксированное число исправно функционирующих вычислителей. Это понятие отражает возможности ВС по обработке информации при наличии заранее определённой структурной избыточности (из ряда вычислителей) и при использовании параллельных программ с указанным числом ветвей.

В теории надёжности ВС, говоря об отказе, понимают событие, при котором система утрачивает способность корректно выполнять функции параллельной программы с указанным числом ветвей. Говорят, что ВС находится в состоянии отказа, если число неисправных вычислителей превышает объём структурной избыточности.

Под живучестью ВС понимается свойство программной настройки и организации функционирования таких структурных схем, которые в условиях возникновения отказов и восстановлений вычислителей гарантируют для выполняющихся параллельных программ производительность в указанных пределах или предоставляют возможность использования всех исправных вычислителей. «Живучесть» ВС характеризует их способность по организации отказоустойчивости при выполнении параллельных программ, способных в

процессе выполнения варьировать количество рабочих ветвей, достигая максимальной производительности.

Выделяют полный и частичный отказы. Полный отказ ВС – это событие, при котором полностью утрачивается способность к выполнению параллельной программы с переменным числом ветвей. Частичный отказ ВС – это событие, при котором возникают отказы отдельных вычислителей системы, однако сохраняется способность к выполнению параллельной программы с переменным числом ветвей. При полном отказе производительность ВС становится равной нулю, в то время как частичный отказ приводит лишь к некоторому снижению данного показателя. После вышесказанного, понятия полного и частичного восстановления становятся очевидными.

Важным моментом живучих ВС является то, что в любой момент времени функционирования системы используется суммарная производительность всех исправных вычислителей. Из этого следует, что программа, реализующая задачу, должна иметь способность адаптироваться к количеству исправно функционирующих вычислителей.

Самоконтроль и самодиагностика (*Self-testing and Self-diagnostics*) ВС. Для обеспечения надёжности и живучести ВС необходимо контролировать правильность работы входящих в её состав вычислителей и при возникновении неисправностей оперативно локализовывать их. Когда речь идёт о системе, основанной на модели коллектива вычислителей, можно применять нетрадиционный подход к контролю и диагностики:

- в качестве контрольно-диагностического ядра ВС могут выступать любые исправные вычислители;
- выбор ядра системы и определение её исправности могут выполнятся с автоматически.

При данном подходе система самостоятельно контролирует правильность работы всех вычислителей и в случае неисправности способна провести самодиагностику. Решение об исправности или неисправности того или иного вычислителя системы принимается всеми вычислителями на основе

индивидуальных заключений об исправности вычислителей соседних с каждым из них.

Технико-экономическая эффективность (*Technical-economical Efficiency*) ВС. Принцип однородности позволяет в значительной степени сократить сроки создания ВС, облегчает процесс её реконфигурации (статической и динамической), оказывает положительное воздействие в процессе эксплуатации. Однородность также способствует процессу организации взаимосвязей между вычислителями ВС и упрощает процесс создания ПО. Три основные принципа модели коллектива вычислителей позволяют создавать высокопроизводительные и экономически приемлемые ВС при имеющейся элементной базе.

Таким образом, описанные выше свойства позволяют классифицировать различные средства обработки информации, производить их анализ и сравнение по различным параметрам.

1.2 Классификация архитектур ВС

В архитектурном плане выделяют четыре класса архитектур вычислительных средств:

1) *SISD* (*Single Instruction stream* / *Single Data stream*);
2) *MISD* (*Multiple Instruction stream* / *Single Data stream*);
3) *SIMD* (*Single Instruction stream* / *Multiple Data stream*);
4) *MIMD* (*Multiple Instruction stream* / *Multiple Data stream*).

SISD архитектура подразумевает такое функционирование, при котором один поток команд управляет обработкой одного потока данных (рисунок 1.1, а); *MISD* – множественный поток команд управляет одним потоком данных (рисунок 1.1, б); *SIMD* – один поток команд управляет множественным потоком данных (рисунок 1.1, в); *MIMD* – множественный поток команд управляет множественным потоком данных (рисунок 1.1, г).

Приведённая классификация была предложена профессором Стенфордского университета США М. Дж. Флинном (*M.J. Flynn*) в 1966 г. и получила широкое распространение.

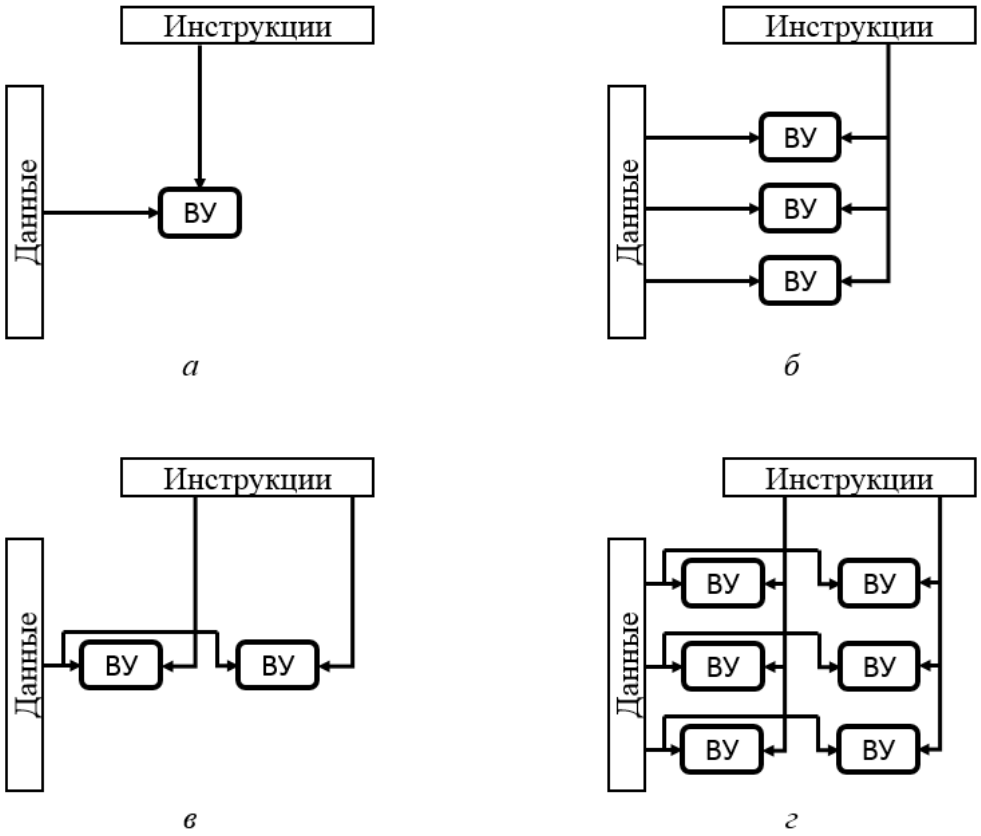

Рисунок 1.1 – Классификация архитектур по Флинну

ВУ – вычислительное устройство

а – SISD; *б – SIMD*; *в – MISD*; *г – MIMD*

Однако, отнести современную ВС только к одному конкретному классу архитектур не представляется возможным, так как на разных уровнях система может проявлять свойства различных классов. Например, ВС в общем виде выполняет множество инструкций над множеством данных (*MIMD*), однако на уровне процессора отдельного вычислителя, наблюдается конвейерная обработка данных (*MISD*). Таким образом, говоря об архитектуре ВС в терминах классификации предложенной М. Дж. Флинном, стоит уточнять уровень, на котором рассматривается система. В [33] описываются следующие типы ВС.

Конвейерные ВС – системы, архитектура которых является предельным вариантом эволюционного развития последовательной ЭВМ. Это простейшая

версия модели коллектива вычислителей. Основой таких ВС являются элементарные блоки обработки информации, соединённые между собой последовательно в виде конвейера. Каждый блок работает параллельно с остальными и реализует лишь собственную операцию над данными одного и того же потока. Данное поведение позволяет отнести конвейерные ВС к классу MISD архитектур.

Матричные ВС – системы, основанные на принципе массового параллелизма, в которых элементарные процессоры объединены в матрицу, обеспечивая возможность одновременного выполнения большого количества операций. Такие ВС, в частности, рассчитаны на решение задач матричной алгебры и, в классическом виде, относятся к классу *SIMD* архитектур. Однако современные высокопроизводительные варианты матричных ВС являются масштабируемыми и относятся к классу *MIMD* архитектур.

Мультипроцессорные ВС – группа систем с *MIMD* архитектурой, состоящая из множества процессоров и общей памяти. Взаимодействие процессора и памяти системы осуществляется по средствам коммутатора (общую шину и т.п.), а между процессорами – через память.

Распределённые ВС – мультипроцессорные ВС, в которых нет общей памяти. Такие ВС основываются на принципах модульности и близкодействия [33] и относятся к классу *MIMD* архитектур.

ВС с программируемой структурой – это композиция взаимосвязанных ЭМ. Такие ВС относятся к классу *MIMD* архитектур, однако их структура может быть программно перестроена соответственно под *MISD* и *SIMD* классы. Данные системы могут функционировать во всех основных режимах: решение сложной задачи, обработка набора задач, обслуживание потока задач.

Кластерные ВС – это композиция множества вычислителей, сети связей между ними и программного обеспечения, предназначенного для параллельной обработки информации. Для создания таких ВС используются все классы рассмотренных выше архитектур, различные функциональные структуры и конструктивные решения. Использование массовых аппаратно-программных

средств в кластерных ВС является основным принципом их конструирования, обеспечивающим их высокую технико-экономическую эффективность.

Суперкомпьютеры – вычислительные средства, имеющие рекордную эффективность (производительность, надёжность, живучесть и технико-экономические показатели) для определённого этапа развития средств обработки информации [9].

Таким образом, классификация Флинна предоставляет базовые понятия архитектур вычислительных средств, однако не позволяет провести однозначное соответствие для современных ВС.

1.3 Надёжность ВС

Под надёжностью (*Reliability*) ВС понимается свойство системы сохранять заданный уровень производительности путём программной настройки её структуры и программной организации функционального взаимодействия между её ресурсами. Данный показатель становится особенно важным, когда речь заходит о распределённых ВС (*Distributed Computer Systems*), количество функциональных элементов в которых может составлять порядка 10^7. Это даёт основание относить распределённые ВС к классу большемасштабных систем (*Large-Scalable Computer System*). Надёжности ВС можно добиться за счёт использования структурной избыточности.

1.3.1 ВС со структурной избыточностью

ВС в общем случае представляют собой совокупность неабсолютно надёжных ЭМ (*N*). В теории функционирования распределённых ВС существуют некоторые показатели, характеризующие работу ЭМ, так λ – интенсивность потока отказов в любой из *N* машин системы. Следовательно, обратная величина λ^{-1} – среднее время безотказной работы одной ЭМ (средняя наработка до отказа) [33].

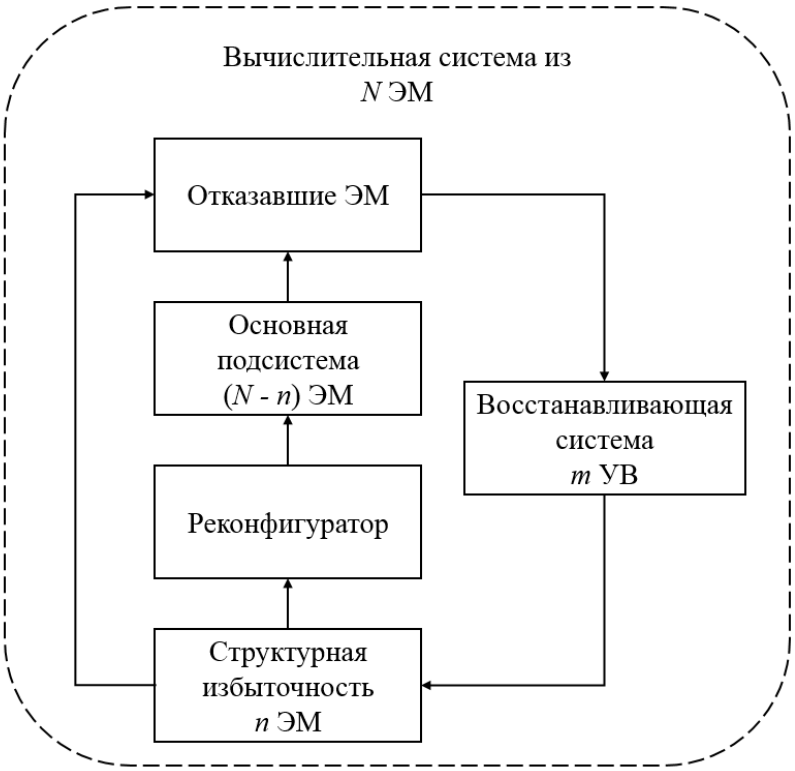

Рисунок 1.2 – Схема функционирования ВС со структурной избыточностью

УВ – устройство восстановления; ЭМ – элементарная машина

Возникающие в ВС отказы устраняются в процессе процедуры восстановления, которая предусматривает контроль функционирования ВС и локализацию имеющихся неисправностей. Данная процедура реализуется при помощи так называемой *восстанавливающей системы* из m *устройств восстановления* (УВ), число которых лежит в пределах от 1 до N. Каждое УВ может быть либо свободным, либо занятым восстановлением не более одной ЭМ. μ – интенсивность восстановления одним УВ.

Таким образом, ВС со структурной избыточностью представляют собой восстанавливающую подсистему из m УВ и совокупность из N взаимосвязанных ЭМ: n – основная подсистема; $N - n$ – структурная избыточность.

На рисунке 1.2 приведена схема функционирования ВС со структурной избыточностью: отказавшая ЭМ поступает в восстанавливающую подсистему; пока УВ занимается устранением неисправности, отказавшую ЭМ заменяет исправная ЭМ из структурной избыточности; после восстановления ЭМ становится частью структурной избыточности. Стоит заметить, что во время функционирования ВС, может отказать любая из N ЭМ.

Реализация ВС со структурной избыточностью является относительно простым и эффективным способом достижения надёжности, особенно если учитывать статистику отказов тех или иных компонентов, используемых ЭМ. Однако, структурная избыточность также вносит и дополнительные расходы.

1.3.2 Показатели надёжности ВС

Для количественного анализа работы ВС используется набор показателей, целью которого является предоставление исчерпывающей информации о возможностях ВС: производительность в текущий момент времени и на некотором его промежутке; способность к восстановлению заданного уровня производительности после отказа отдельных элементов системы; поведение системы на начальном этапе её работы (*переходный режим*) и при длительной эксплуатации (*стационарный режим*) [33, 21]. Далее рассматриваются эти показатели более детально.

Функция надёжности – вероятность того, что производительность ВС, начавшей функционировать в состоянии i ($n \le i \le N$) на промежутке времени [0, t), равна производительности основной подсистемы. Формальные эквивалентные записи определения функции надёжности:

$$R(t) = P\{\forall\tau \in [0,t) \to \Omega(t) = A_n n\omega \mid n \le i \le N\};$$

$$R(t) = P\{\forall\tau \in [0,t) \to \xi(t) \ge n \mid n \le i \le N\},$$

где A_n – коэффициент производительности для n-ой ЭМ; ω – производительность отдельной ЭМ; $\Omega(\tau)$ – производительность системы в момент времени τ; $\xi(\tau)$ – количество исправных ЭМ в системе в момент времени τ.

Функция восстановимости – вероятность того, что в ВС, имеющей начальное состояние i ($0 \le i \le n$), будет восстановлен на промежутке времени [0, t) уровень производительности, равный производительности основной подсистемы. Формальные эквивалентные записи определения функции восстановимости:

$$U(t) = 1 - P\{\forall\tau \in [0,t) \to \Omega(t) = 0 \mid 0 \ \le i < n\};$$

$$U(t) = 1 - P\{\forall\tau \in [0,t) \to \xi(t) < n \mid 0 \le i < n\}.$$

На практике чаще используются математическое ожидание времени безотказной работы и среднее время восстановления ВС, которые соответственно равны:

$$\Theta = \int_0^\infty R(t)dt;$$

$$T = \int_0^\infty tdU(t).$$

Функция готовности – вероятность того, что производительность системы, начавшей функционировать в состоянии i $(0 \le i \le N)$, равна в момент времени $t \ge 0$ производительности основной подсистемы:

$$S(t) = P\{\xi(t) \ge n \mid 0 \le i \le N\}.$$

Описанные выше показатели отражают характер поведения ВС и её способность к восстановлению в случае возникновения отказов лишь на начальном этапе функционирования (переходный режим). Для стационарного режима вводятся *функции оперативной надёжности и восстановимости ВС*.

Функция оперативной надёжности – вероятность того, что производительность системы, которая в начальный момент находится в состоянии i $(n \le i \le N)$, с вероятностью P_i, равна на промежутке времени $[0, t)$ производительности основной подсистемы. Формальные эквивалентные записи определения функции оперативной надёжности:

$$R^*(t) = P\{\forall\tau \in [0,t) \to \Omega(t) = A_n n\omega \mid P_i,\ n \le i \le N\};$$

$$R^*(t) = P\{\forall\tau \in [0,t) \to \xi(t) \ge n \mid\ P_i,\ n \le i \le N\}.$$

Функция оперативной восстановимости – вероятность того, что в ВС, находящейся в начальном состоянии i $(0 \le i < n)$, с вероятностью P_i, равна на промежутке времени $[0,\ t)$ будет восстановлен уровень производительности основной подсистемы. Формальные эквивалентные записи определения функции оперативной восстановимости:

$$U^*(t) = 1 - P\{\forall\tau \in [0,t) \to \Omega(t) = 0 \mid P_i,\ 0 \le i < n\};$$

$$U^*(t) = 1 - P\{\forall\tau \in [0,t) \to \xi(t) < n \mid P_i,\ 0 \le i < n\}.$$

В отличии от функций надёжности и восстановимости функция готовности может быть использована в стационарном режиме и её значении в пределе не зависит от начального состояния системы. Данный показатель называется *коэффициентом готовности ВС* (S):

$$\lim_{t\to\infty} S(t) = \sum_{j=n}^{N} \lim_{t\to\infty} P_j(i,t) = \sum_{j=n}^{N} P_j = S,$$

где $P_j(i,t)$ – вероятность того, что в системе, начавшей функционировать в состоянии i ($0 \le i \le N$), в момент $t \ge 0$ будет j ($0 \le j \le N$) исправных машин.

Таким образом функции надёжности и готовности характеризуют возможности ВС обеспечить определённый уровень производительности на промежутке времени [0, t) и в момент времени $t \ge 0$. Функция восстановимости отражает возможности ВС к восстановлению указанного уровня производительности после возникновения отказа.

Данный набор показателей позволяет производить объективную оценку конкретной ВС, способствую улучшению качества её функционирования, и делать прогнозы возможных отказов и сбоев в работе ВС.

1.4 Живучесть ВС

Под живучестью (*Robustness*) понимается способность ВС в любой момент функционирования использовать суммарную производительность всех исправных ЭМ [33]. Живучесть достигается за счёт программной организации структуры ВС и функционального взаимодействия её компонентов. Данное свойство должно проявляться при решении задач как в монопрограммном, так и в мультипрограммных режимах функционирования ВС.

1.4.1 Живучие ВС

Живучая ВС – программная конфигурация из N ЭМ, в которой:

1) требуемый уровень производительности достигается за счёт заранее указанного количества n работоспособных ЭМ;

2) имеется возможность выполнения адаптирующихся параллельных программ;
3) отказы и восстановления любых ЭМ приводят только лишь к увеличению или уменьшению времени реализации параллельной программы соответственно.

Производительность таких ВС можно описать выражением

$$\Omega(k) = A_k \Delta(k-n)\varphi(k,\omega);$$

$$\Delta(k-n) = \begin{cases} 1, \text{если } k \geq n; \\ 0, \text{если } k < n, \end{cases}$$

где k $(0 \leq k \leq N)$ – текущее состояние ВС; A_k – коэффициент производительности; $\varphi(k,\omega)$ – неубывающая функция от k и ω (как правило, $\varphi(k,\omega) = k\omega$ при решении сложных задач).

Для формирования вычислительного ядра в живучей ВС имеется так называемый *реконфигуратор*, который исключает из вычислительного ядра отказавшие ЭМ и включает в него восстановленные; занимается процессом вложения параллельной программы в вычислительное ядро.

На рисунке 1.3 представлены графики зависимости производительности ВС со структурной избыточностью и живучей ВС от количества исправных ЭМ в них.

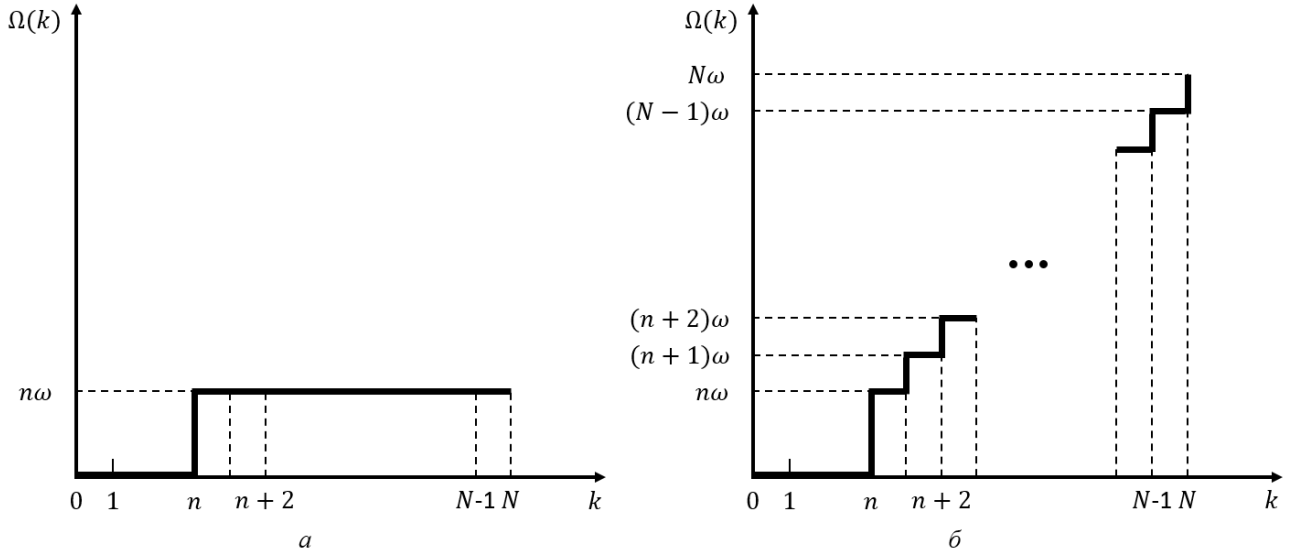

Рисунок 1.3 – Производительность вычислительных систем

а – ВС со структурной избыточностью; *б* – живучие ВС

Таким образом, в живучих ВС ЭМ, составляющие структурную избыточность, не простаивают и отказ одной из ЭМ не приводит к отказу системы в целом, а лишь оказывает некоторое влияние на её производительность.

1.4.2 Показатели потенциальной живучести ВС

Для описания характера работы живучих ВС существует некоторый набор показателей, к которому, как и в случае показателей надёжности, предъявляются определённые требования: необходимость учитывать непостоянное количество ЭМ, участвующих в процессе решения задач [33].

Функция потенциальной живучести $\mathcal{N}(i,t)$ характеризует в момент времени $t \geq 0$ среднюю производительность системы, при условии что на момент начала функционирования в ВС было i $(0 \leq i \leq N)$ исправных ЭМ. Формальная запись определения:

$$\mathcal{N}(i,t) = \frac{\overline{\Omega}(i,t)}{N\omega},$$

где $\overline{\Omega}(i,t)$ – математическое ожидание производительности ВС в момент времени $t \geq 0$, при условии что на момент начала функционирования в ВС было i $(0 \leq i \leq N)$ исправных ЭМ; $N\omega$ – суммарная производительность всех ЭМ; N – общее количество ЭМ в системе; ω – производительность отдельной ЭМ.

Функция занятости восстанавливающей системы $\mathcal{M}(i,t)$ характеризует в момент времени $t \geq 0$ среднюю загруженность восстанавливающей системы, при условии что на момент начала функционирования в ВС было i $(0 \leq i \leq N)$ исправных ЭМ. Формальная запись определения:

$$\mathcal{M}(i,t) = \frac{m(i,t)}{m},$$

где $\overline{\Omega}(i,t)$ – математическое ожидание числа занятых восстанавливающих устройств в момент времени $t \geq 0$, при условии что на момент начала функционирования в ВС было i $(0 \leq i \leq N)$ исправных ЭМ; m – количество устройств в восстанавливающей системе.

В набор показателей также входят *вектор-функции* $\boldsymbol{R}(t), \boldsymbol{U}(t), \boldsymbol{S}(t)$, являющиеся обобщениями функций надёжности, восстановимости и готовности ВС.

Вектор-функция надёжности

$$\boldsymbol{R}(t) = \{R_k(t)\}, n \le k \le N,$$

где $R_k(t)$ – вероятность того, что производительность системы, начавшей функционировать в состоянии i $(0 \le i \le N)$, не менее производительности k ЭМ на всём промежутке времени $[0, t)$. Формальные эквивалентные записи:

$$R_k(t) = P\{\forall\tau \in [0,t) \to \Omega(t) = A_k k\omega \mid k \le i \le N\};$$

$$R_k(t) = P\{\forall\tau \in [0,t) \to \xi(t) \ge k \mid k \le i \le N\},$$

где $\Omega(t)$ и $\xi(t)$ – производительность ВС и количество исправных ЭМ в системе в момент времени $\tau \in [0,t)$ соответственно; i – начальное состояние ВС.

Вектор-функция восстановимости

$$\boldsymbol{U}(t) = \{U_k(t)\}, n \le k \le N,$$

где $U_k(t)$ – вероятность того, что в системы, начавшей функционировать в состоянии i $(0 \le i \le N)$, на промежутке времени $[0, t)$ будет восстановлен уровень производительности не менее k ЭМ. Формальные эквивалентные записи:

$$U_k(t) = 1 - P\{\forall\tau \in [0,t) \to \Omega(t) < A_k k\omega \mid 0 \le i < k\};$$

$$U_k(t) = 1 - P\{\forall\tau \in [0,t) \to \xi(t) < k \mid 0 \le i < k\}.$$

Можно также рассматривать и *вектор среднего времени безотказной работы*

$$\boldsymbol{\varkappa} = \{\Theta_k\}, \qquad \Theta_k = \int_0^\infty R_k(t)dt$$

и *вектор среднего времени восстановления*

$$\mathbf{T} = \{T_k\}, \qquad T_k = \int_0^\infty t dU_k(t).$$

Вектор-функция готовности

$$\boldsymbol{S}(t) = \{S_k(t)\}, n \le k \le N,$$

где $S_k(t)$ – вероятность того, что в момент времени $t \geq 0$ производительность системы будет не менее k ЭМ, при условии что на момент начала функционирования в ВС было $i\ (0 \leq i \leq N)$ исправных ЭМ. Формальные эквивалентные записи:

$$S_k(t) = P\{\Omega(t) \geq A_k k\omega \mid 0 \leq i \leq N\};$$

$$S_k(t) = P\{\xi(t) \geq k \mid 0 \leq i \leq N\}.$$

Вышеприведённые показатели применимы в переходном режиме функционирования ВС. Для стационарного режима необходимо рассматривать предельные значения. Так, функции потенциальной живучести и занятости восстанавливающей системы при длительной эксплуатации ВС называются *коэффициентом потенциальной живучести* и *коэффициентом занятости восстанавливающей системы* соответственно и не зависят от начального состояния i:

$$\mathcal{N} = \lim_{t\to\infty} \mathcal{N}(i,t)\,;$$

$$\mathcal{M} = \lim_{t\to\infty} \mathcal{M}(i,t).$$

Аналогично вектор-функции переходного режима имеют соответствующие показатели для стационарного режима работы ВС.

Вектор-функция оперативной надёжности

$$\boldsymbol{R}^*(t) = \{{R^*}_k(t)\}, n \leq k \leq N,$$

где ${R^*}_k(t)$ – вероятность того, что в системы, начавшей функционировать в состоянии $i\ (0 \leq i \leq N)$, на промежутке времени [0, t) с вероятностью P_i будет восстановлен уровень производительности не менее k ЭМ. Формальные эквивалентные записи:

$${R_k}^*(t) = P\{\forall\tau \in [0,t) \to \Omega(t) = A_k k\omega \mid P_i,\ k \leq i \leq N\};$$

$${R_k}^*(t) = P\{\forall\tau \in [0,t) \to \xi(t) \geq k \mid\ P_i,\ k \leq i \leq N\}.$$

Вектор-функция оперативной восстановимости

$$\boldsymbol{U}^*(t) = \{{U^*}_k(t)\}, n \leq k \leq N,$$

где ${U^*}_k(t)$ – вероятность того, что в момент времени $t \geq 0$ производительность системы с вероятность P_i будет не менее k ЭМ, при условии что на момент начала

функционирования в ВС было $i\,(0 \le i \le N)$ исправных ЭМ. Формальные эквивалентные записи:

$$U_k{}^*(t) = 1 - P\{\forall\tau \in [0,t) \to \Omega(t) < A_k k\omega \mid P_i,\ 0 \le i < k\};$$

$$U_k{}^*(t) = 1 - P\{\forall\tau \in [0,t) \to \xi(t) < k \mid P_i,\ 0 \le i < k\}.$$

Совокупность величин $S_k = \lim\limits_{t\to\infty} S_k(t)$, не зависящих от начального состояния ВС i и представленных в виде

$$\boldsymbol{S} = \{S_k(t)\}, n \le k \le N,$$

называется *вектор-коэффициентом готовности*.

Описанный набор показателей достаточно полно описывает поведение живучих ВС в обоих режимах функционирования ВС (переходном и стационарном) и позволяет определить: скорость перехода ВС в стационарный режим работы; среднюю производительность системы в любой момент времени или на некотором его промежутке; уровень загрузки восстанавливающих устройств; оптимальные параметры единого комплекса «вычислительная система – восстанавливающая система». **Параллельные алгоритмы и программы**

Алгоритм (алгорифм) (лат. *Algorithmi* – имя среднеазиатского математика аль-Хорезми) – точное предписание о выполнении в определённом порядке некоторой системы операций, ведущих к решению всех задач данного типа. Простейшими примерами алгоритма являются арифметические правила сложения, вычитания, умножения и деления, правила извлечения квадратного корня, способ нахождения общего наибольшего делителя для двух любых натуральных чисел и др. По существу, с алгоритмом мы имеем дело всегда, когда обладаем средствами решать ту или иную задачу в общем виде, то есть для целого класса её варьируемых условий. Поскольку алгоритм, кик система предписаний, носит формальный характер на основе всегда можно разработать программу действий для вычислительной машины и осуществить машинное решение задачи. Выявление алгоритма решения широкого круга задач и разработка теории алгоритма особенно актуальна в связи с развитием вычислительной техники и кибернетики [32]. Опираясь на определения

коллектива вычислителей и алгоритма можно сказать, что такое параллельный алгоритм. *Параллельный алгоритм* – это точное предписание о выполнении в определённом порядке некоторой системы операций, ориентированных на реализацию в коллективе вычислителей и ведущих к решению всех задач данного типа. Такой алгоритм в отличии от последовательного предусматривает одновременное выполнение нескольких операций за один шаг вычислений.

Запись на языке программирования действий, основанной на параллельном алгоритме решения задачи, для вычислительной машины называется *параллельной программой* (*Parallel Program*), а сам язык – *параллельным* (*Parallel Language*). Параллельные алгоритмы и программы следует разрабатывать для трудоёмких задач.

Процесс приведения последовательной версии алгоритма решения сложных задач к параллельной называется *распараллеливанием* (*Paralleling or Multisequencing*).

Деятельность, связанная с созданием параллельных алгоритмов и программ для решения различных задач, называется *параллельным программированием* (*Parallel or Concurrent Programming*).

1.4.3 Методики распараллеливания сложных задач

Существуют два подхода к распараллеливанию сложных задач: локальное и глобальное (крупноблочное) распараллеливание (рисунок 1.4). Первый подход базируется на разбиении входной задачи на максимально простые блоки и выделении для каждого этапа вычислений предельно возможного количества параллельно выполняемых блоков. При данном подходе в модели коллектива вычислителей выполняется наибольшее количество информационных обменов между модулями системы. Помимо большого количества обменов, появляется высокая вероятность неравномерности в объёмах, выполняемых каждым вычислителем операций. С учётом вышесказанного и того, что в большемасштабных ВС межмашинные связи являются «узким местом»,

локальное распараллеливание является трудоёмким процессом и позволяет получить придельные оценки по распараллеливанию сложных задач.

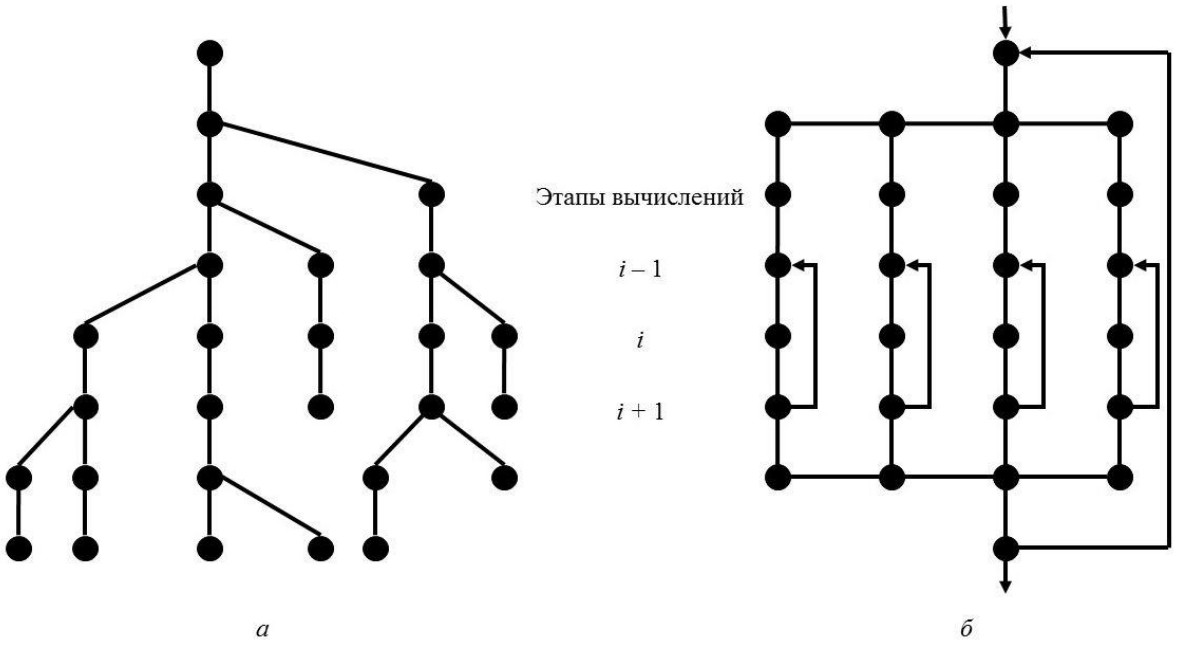

Рисунок 1.4 – Фрагменты схемы параллельных алгоритмов:
а – локальное распараллеливание; *б* – глобальное распараллеливание

Второй подход основан на разбиении входной задачи на достаточно большие (крупные) подзадачи (блоки). При таком подходе количество обменов между вычислителями минимально, что положительно сказывается на времени выполнения всей задачи, благодаря эффективному использованию архитектурных возможностей ВС. Подзадачи называют ветвями параллельного алгоритма, а соответствующие им программы – ветвями параллельной программы.

Распространённым приёмом крупноблочного распараллеливания сложных задач является распараллеливания циклов. Это позволяет разделить процесс решения на независимые (как правило) параллельные ветви, которые получаются за счёт расщепления цикла на части. Каждая ветвь выполняет последовательный набор операций. Процесс решения задачи с применением данного приёма начинается инициализацией ветвей, а заканчивается синтезом их результатов.

В настоящее время на практике можно встретить оба подхода, однако крупноблочное программирование наиболее популярно. Для демонстрации рассмотрим алгоритм решения системы линейных алгебраических уравнений (СЛАУ) методом Гаусса.

1.4.4 Показатели эффективности параллельных алгоритмов

Как уже указывалось ранее, параллельный алгоритм состоит из связанных между собой ветвей. Очевидно, что при их взаимодействии возникают накладные расходы, следовательно, чем реже происходят информационные обмены, тем эффективнее работает параллельный алгоритм.

Таким образом, можно ввести первый показатель эффективности – *коэффициент накладных расходов* [18]

$$\varepsilon = \frac{t}{T},$$

где t – время всех информационных обменов между ветвями; T – время выполнения вычислений при реализации параллельного алгоритма.

Следующий показатель – *коэффициент ускорения* [24], который для параллельного алгоритма вычисляется по формуле

$$\chi = \frac{\tau_1}{\tau_n},$$

где τ_1 – время решения задачи на одном вычислителе; τ_n – время решения задачи на ВС с n активными вычислителями.

Также выделяют коэффициент *ускорения параллельного алгоритма по сравнению с наилучшим последовательным алгоритмом*

$$\chi' = \frac{\tau'_1}{\tau_n},$$

здесь τ'_n – время работы самой быстрой версии последовательного алгоритма.

Существует *закон Амдала*, который позволяет оценить насколько метод решения сложной задачи приспособлен к параллельному представлению. Ускорение, которое может быть получено на ВС из n активных вычислителей, по сравнению с последовательным решением не будет превышать величины

$$S_n = \frac{1}{\alpha + \frac{1-\alpha}{n}},$$

где α – относительная доля параллельной программы, выполняемая последовательно, $0 \leq \alpha \leq 1$.

Помимо этого, в параллельном программировании используют *коэффициент эффективности*

$$E = \frac{\chi}{n}$$

и *коэффициент эффективности параллельной версии алгоритма по отношении к наилучшей последовательной версии*

$$E' = \frac{\chi'}{n}.$$

Как правило коэффициент эффективности меньше или равен единице $(E \leq 1)$, однако могут возникать ситуации («парадокс» параллелизма), при которых его значение превышает указанные рамки. Это связанно с особенностями организации ВС и работы её отдельных элементов, например, памяти вычислителей.

Одной из целей параллельного программирования является достижение максимального ускорения ($\chi = n; E = 1$), но добиться этого зачастую не удаётся по ряду причин:

- наличие накладных расходов на синхронизацию параллельных ветвей;
- возникновение «конфликтов» памяти [11, 14-16, 22] (особенности её работы);
- несбалансированность нагрузки вычислителей.

Таким образом, показатели эффективности позволяют давать объективную оценку различным параллельным алгоритмам в определённых рамках.

1.4.5 Решение СЛАУ методом Гаусса

Метод Гаусса является классическим способом решения СЛАУ [19].

Пусть исходная система имеет следующий вид:

$$\begin{cases} a_{11}x_1 + \cdots + a_{1n}x_n = b_1 \\ \cdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n = b_m \end{cases}, \qquad (2.1)$$

где a_{ij} – коэффициенты при переменных x_j $(1 \le i \le m; 1 \le j \le n)$; x_j – главные переменные; b_i – свободные члены.

Система 2.1 также представима в матричном виде:

$$A\bar{x} = \bar{b},$$

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ & \dots & \\ a_{m1} & \dots & a_{mn} \end{pmatrix}, \quad \bar{x} = \begin{pmatrix} x_1 \\ . \\ . \\ . \\ x_m \end{pmatrix}, \quad \bar{b} = \begin{pmatrix} b_1 \\ . \\ . \\ . \\ b_m \end{pmatrix}.$$

Согласно свойству элементарных преобразований над строками, которое гласит, что при выполнении перестановки местами любых двух строк матрицы или умножении любой её строки на константу $k, k \neq 0$, или прибавлении к любой строке этой матрицы другой её строки, умноженной на некоторую константу, сохраняется эквивалентность матриц. Тогда основную матрицу A можно привести к нижнетреугольному виду (эти преобразования так же должны применяться и к столбцу свободных членов $\bar{b}$):

$$A' = \begin{pmatrix} \alpha_{11} & \alpha_{12} & \dots & \alpha_{1n} \\ 0 & \alpha_{22} & \dots & \alpha_{2n} \\ & . \ . & . \ . & \\ 0 & 0 & \dots & \alpha_{mn} \end{pmatrix}, \quad \bar{x} = \begin{pmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{pmatrix}, \quad \bar{b}' = \begin{pmatrix} \beta_1 \\ . \\ . \\ . \\ \beta_m \end{pmatrix}.$$

После выполнения вышеперечисленных действий, становится возможным нахождение значений главных переменных $\bar{x}$, начиная с x_m и заканчивая x_1. Так, например, значение x_m определяется выражением

$$x_m = \frac{\beta_m}{\alpha_{mn}}.$$

Аналогично для остальных значений столбца $\bar{x}$:

$$x_j = (\beta_j - \sum_{l=j+1}^{n} \alpha_{jl} x_l) \ / \ \alpha_{jj}. \qquad (2.2)$$

Стоит заметить, что данный метод применим только, если $i \ge j$.

Последовательная версия алгоритма решения СЛАУ методом Гаусса. В данной версии подразумевается, что только один процесс принимает участие

в вычислении. Также стоит заметить, что индексирование элементов матрицы начинается с нуля. Алгоритм делится на два этапа:

1) Преобразование основной матрицы A к нижнетреугольному виду (*прямой проход*).
2) Нахождение значений главных переменных $\bar{x}$ (*обратный проход*).

Листинг 1.1 – Псевдокод последовательной версии алгоритма решения СЛАУ методом Гаусса

```
1    Input n
2    Input A[n][n+1]
3    // Прямой проход
4    For i = 0; i < n ; i = i + 1
5        K = 1.0 / A[i][i]
6        For j = i; j < n + 1; j = j + 1
7            A[i][j] = A[i][j] * K
8        For j = i + 1; j < n; j = j + 1
9            K = -1.0 * A[j][i]
10           For l = i; l < n + 1; l = l + 1
11               A[j][l] = A[j][l] + A[i][l] * K
12   // Обратный проход
13   For i = n - 1; i ≥ 0; i = i - 1
14       X[i] = A[i][n]
15       For j = 1; j < n - i; j = j + 1
16           X[i] = X[i] - X[i-j] * A[n - j]
17   Output X
```

В листинге 1.1 представлен псевдокод последовательной версии алгоритма решения СЛАУ методом Гаусса. В строках 1 и 2 вводится размер основной матрицы и заполняется её расширенный вариант соответственно (подразумевается отсутствие нулевых значений в основной матрице). Строки 4-11 содержат инструкции первого этапа алгоритма – *прямого прохода*. В 5 строке вычисляется коэффициент для приведения текущей рабочей строки (относительно которой выполняются элементарные преобразования) расширенной матрицы к виду с единичным значением диагонального элемента

α_{ii}=1. Инструкции в 6 и 7 строках выполняют описанное выше преобразование. Далее (строки 8-11), происходит обработка оставшихся $n-i$ строк, таким образом, чтобы элементы *i*-ого столбца были равны нулю. После первого этапа, матрица имеет нижнетреугольный вид, а её диагональ состоит из единичных элементов. Строки 13-16 описывают обратный проход, при котором определяются значения главных переменных $\bar{x}$. Как указывалось выше, диагональ получившейся матрицы имеет единичный вид $\alpha_{jj}=1$, следовательно, для нахождения значения *i*-ой переменной необходимо из соответствующего свободного члена вычесть сумму всех элементов обрабатываемой строки матрицы умноженных на уже известные значения главных переменных (формула 2.2). В 17 строке выводится результат работы алгоритма – значения главных переменных. Данный алгоритм прост в реализации и приводим к параллельному виду.

Параллельная версия алгоритма решения СЛАУ методом Гаусса. При параллельной обработки информации входная задача разбивается на части, которые выполняются одновременно *r* процессами. Существует ни один способ организации параллельного решения СЛАУ методом Гаусса. В данной работе рассматривается вариант, при котором входная матрица хранится в памяти каждого процесса $P_u(0 \le u < r)$, а распределение рабочих строк на этапе прямого прохода осуществляется согласно формуле 2.3 для всех процессов кроме последнего и 2.4 в противном случае:

$$S_u = \frac{n-(i+1)}{r}; \tag{2.3}$$

$$S_u = \frac{n-(i+1)}{r} + (n-(i+1))\ mod\ r, \tag{2.4}$$

где S_u – количество обрабатываемых *u*-ым процессом строк; n – количество строк в матрице; *r* – количество параллельных процессов; i $(0 \le i < n)$ – номер текущей рабочей строки (шаг прямого прохода); *mod* – операция деления с остатком.

Таким образом, на каждом шаге прямого прохода u-ый процесс будет обрабатывать строки с $start_u = i + 1 + S_u * u$ по $end_u = start_u + S_u$, за исключением последнего процесса, который должен учитывать возможный «остаток». Номера его рабочих строк лежат в границах с $start_u = i + S_{u-1} * u$ по $end_u = start_u + S_u$.

Листинг 1.2 – Псевдокод прямого прохода параллельной версии алгоритма решения СЛАУ методом Гаусса

```
1    Function Gauss_Forward(matrix A, size n)
2        // Преобразование рабочей строки
3        For ROW = 0; ROW < n; ROW = ROW + 1
4            K = A[ROW][ROW]
5            For i = ROW + 1; i ≤ n; i = i + 1
6                A[ROW][i] = A[ROW][i] / K
7            A[ROW][ROW] = 1.00
8            // Если рабочая строка является последней в матрице A
9            If ROW = n - 1
10               Break
11           SIZE = (n – (ROW + 1)) / R
12           START = ROW + 1 + SIZE * RANK
13           // Если остались нераспределённые строки
14           If RANK = R – 1
15               SIZE = SIZE + (n – (ROW + 1)) mod R
16           // Обработка каждым процессом указанной части матрицы
17           For i = 0; i < SIZE; i = i + 1
18               K = 0.00 – A[START + i][ROW]
19               For j = ROW + 1; j ≤ n; j = j + 1
20                   A[START + i][j] = A[START + i][j] +
21                                     A[ROW][j] * K
22               A[START + i][ROW] = 0.00
23           // Обмен результатами вычислений
24           // между каждым процессом
25           SIZE = (n – (ROW + 1)) / R
26           For i = 0; i < R -1; i = i + 1
27               START = ROW + 1 + SIZE * i
```

```
28                Broadcast(A[START], SIZE, i)
29            // Рассылка последним процессом
30            SIZE = SIZE + (n – (ROW + 1)) mod R
31            START = ROW + 1 + SIZE * (R – 1)
32            Broadcast(A[START], SIZE, R - 1)
```

В листинге 1.2 представлен псевдокод прямого прохода параллельной версии алгоритма решения СЛАУ методом Гаусса. 3-7 строки описывают процесс преобразования рабочей строки к необходимому виду: диагональный элемент должен равняться единице α_{ii}=1 (*ROW* – индекс рабочей строки, *K* – коэффициент преобразования). Далее каждый процесс определяет количество обрабатываемых им строк *SIZE* и индекс строки (в рамках всей матрицы) *START*, с которой ему необходимо начать обработку (*RANK* – номер процесса; *R* – общее количество параллельных процессов). После выполнения каждым процессом необходимых действий по приведению входной матрицы к нижнетреугольному виду, промежуточный результат рассылается широковещательным сообщением все другим процессам и формируется результирующая матрица (строки 25-28). Стоит заметить, что отправка сообщения последним процессом (строки 30-32), обрабатывается отдельно, так как количество входящих в его рабочее поле строк может отличаться от аналогичного значения в остальных процессах, следовательно, объём передаваемых данных будет так же другим. На следующем шаге прямого прохода, рабочей строкой становится ниже идущая и описанные выше действия повторяются (у каждого процесса будет новая расчётная область). В итоге после прямого прохода матрица принимает нижнетреугольный вид, с единичной диагональю.

При обратном проходе возникает информационная избыточность (дублирование вычислений), что положительно сказывается на надёжности, но негативно на времени выполнения задачи: каждый процесс на данном этапе хранит в своей памяти матрицу, приведённую к нижнетреугольному виду,

следовательно, этого достаточно, чтобы независимо от других процессов, определить значения главных переменных.

Листинг 1.3 – Псевдокод обратного прохода параллельной версии алгоритма решения СЛАУ методом Гаусса

```
1    Function Gauss_Backward(matrix A, size n)
2        For i = n – 1; i ≥ 0; i = i – 1
3            X[i] = A[i][n]
4            For j = 1; j < n – i; j = j + 1
5                X[i] = X[i] – X[i-j] * A[n – j]
6        Return X
```

Как видно из листингов 1.1 и 1.3 псевдокод обратного прохода последовательной версии алгоритма совпадает с псевдокодом аналогичного этапа параллельной версии. Процесс нахождение значений главных переменных не подвергается распараллеливанию по той причине, что здесь имеет место зависимость по данным, которая в нашем случае, сводит на нет все преимущества параллельных вычислений.

Такой способ организации параллельного выполнения рассматриваемого алгоритма выбран специально, для его дальнейшей модификации в живучую версию.

1.5 Схемы обмена информацией

Понятие параллельный алгоритм непосредственно связано с информационными обменами между его ветвями. В теории ВС выделяют *дифференцированные* (ДО), *трансляционные* (ТО), *трансляционно-циклические* (ТЦО), *конвейерно-параллельные* (КПО) и *коллекторные* (КО) *обмены* [33].

Дифференцированный обмен является элементарным и осуществляет передачу информации от одной ветви параллельного алгоритма в любую другую (рисунок 1.5, а). Очевидно, что в таких обменах участвуют только два вычислителя – приёмник и передатчик. Следовательно, остальные ресурсы ВС простаивают и это приводит к их неэффективному использованию. Стоит

отметить, что подразумевается ориентация параллельного алгоритма на все доступные вычислители в системе.

Трансляционный обмен (*One-to-all Broadcast*) осуществляет передачу одной и той же информации из одной ветви параллельного алгоритма одновременно во все остальные его ветви (рисунок 1.5, б). При таком виде обменов ресурсы ВС используются эффективно – участвуют все вычислители. Вообще, эффективность использования ресурсов ВС при групповых обменах очевидна.

Трансляционно-циклический обмен (*All-to-all Broadcast*) обеспечивает передачу информации из каждой ветви во все остальные. Данный вид обмена выполняется за *n* тактов, в отличии от трансляционного обмена, который выполняется за 1 такт.

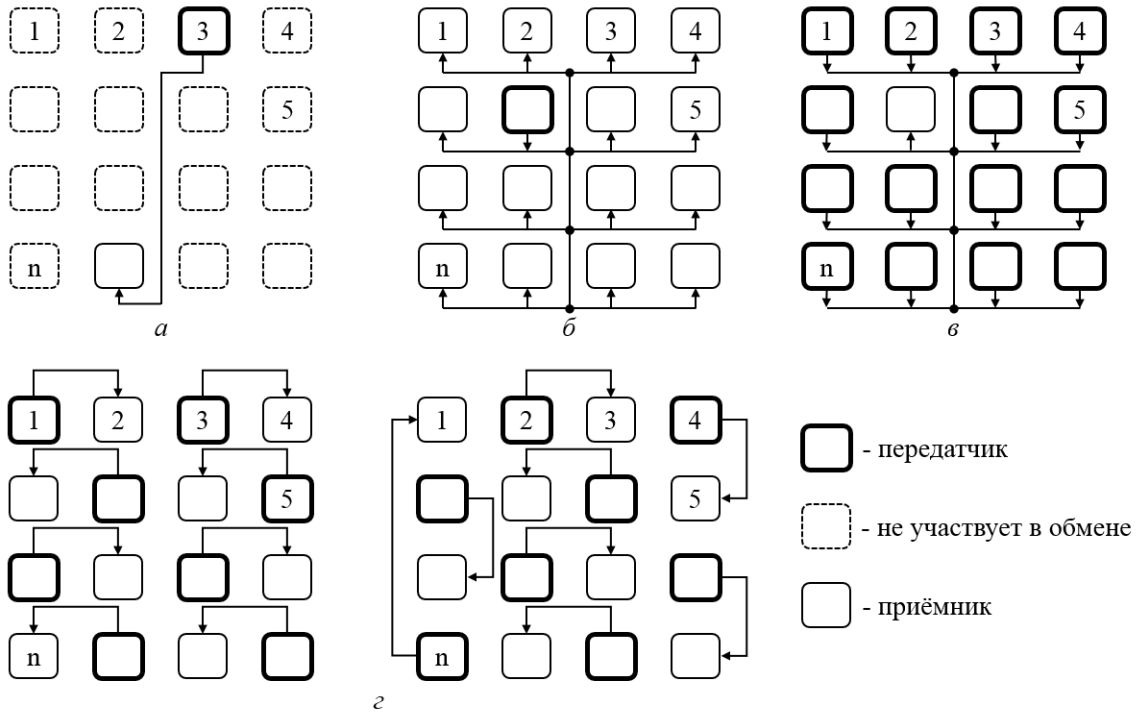

Рисунок 1.5 – Виды информационных обменов в параллельных алгоритмах:
а – ДО; б – ТО; в – КО; г – КПО

Конвейерно-параллельный обмен реализует передачу информации между соседними ветвями параллельного алгоритма. Данный вид обмена выполняется за два такта (рисунок 1.5, г). При чётном *n* на первом такте информация

передаётся из ветвей $P_1, P_3, \dots, P_{n-3}, P_{n-1}$ соответственно в ветви $P_2, P_4, \dots, P_{n-2}, P_n$. На втором такте – из $P_2, P_4, \dots, P_{n-2}, P_n$ в $P_3, P_5, \dots, P_{n-1}, P_1$.

Коллекторный обмен является инвертированным трансляционным обменом: вычислители-приёмники становятся передатчиками, а вычислитель-передатчик – приёмником (рисунок 1.5, в). Такой вид обмена реализован как последовательность из дифференцированных обменов.

Согласно данным из [33], групповые схемы информационных обменов (ТО, ТЦО, КПО) между ветвями параллельных алгоритмов или программ, составленных опираясь на методику крупноблочного программирования, составляют более 90% от общего количества обменов. Таким образом, достигается эффективное использование ресурсов ВС.

Резюмируя, описанные выше схемы информационных обменов являются фундаментальными в области параллельного программирования, основанного на передачи сообщений.

1.6 Средства параллельного программирования

Для разработки параллельных программ существует масса разнообразных инструментальных средств [20]. В системах с общей памятью принято использовать средства многопоточного программирования: *POSIX Threads*, *OpenMP*, *Cilk*/*Cilk++*, *Intel TBB*, *OpenCL*, *NVIDIA CUDA*. Когда речь идёт о системах с распределённой памятью, говорят о средствах, основанных на принципе передачи сообщений: *MPI, SHMEM*, *Unified Parallel C*.

Стандарт *MPI* получил широкое распространение на этапе развития большемасштабных ВС и занимает лидирующие позиции на сегодняшний день.

1.6.1 Message Passing Interface

Message Passing Interface (***MPI***, интерфейс передачи сообщений) – это стандарт интерфейса библиотеки передачи сообщений. *MPI* в первую очередь ориентирован на использование в моделях параллельного программирования, в которых данные передаются из адресного пространства одного процесса в пространство другого по средствам совместных операций, проводимых для

каждого из задействованных процессов. Расширение «классической» модели передачи сообщений достигается за счёт коллективных операций, операций доступа к удалённой памяти, динамического создание процессов и средств параллельного ввода/вывода.

Стоит понимать, что *MPI* – это спецификация, а не реализация, которых существует достаточное количество. Две самые популярные реализации – это *MPICH* и *OpenMPI*. Каждая из них имеет свои особенности, однако любая реализация должна удовлетворять ряду требований, зависящих от версии стандарта (на момент написания этой работы актуальной версией являлась *MP*I 3.0). Подробности о стандарте 3.0 можно найти в соответствующей документации [6].

Спецификация *MPI* предназначена для библиотечного интерфейса; *MPI* не является языком программирования, и все *MPI* операции представляются как функции, подпрограммы или методы, ассоциируемые с определёнными языковыми связками на языках *C* и *Fortran*, которые составляют часть стандарта.

Главным достоинством *MPI* можно считать его портативность и относительную простоту в использовании. В распределённых вычислительных системах высокоуровневые процедуры и/или некоторые абстракции состоят из наиболее простых процедур передачи сообщений, что делает преимущества стандартизации ещё более очевидными.

Цель интерфейса передачи сообщений, в первую очередь, предоставить разработчикам широко используемы стандарт для написания *MPI*-программ. Далее приводится полный список целей: параллельных ветвей программы;

1) Разработка интерфейса прикладного программирования.
2) Организация эффективных средств коммуникации.
3) Разработка реализаций, которые можно было бы использовать в гетерогенных средах.
4) Создание удобных языковых связок для *C* и *Fortran*.
5) Разработка надёжного коммуникационного интерфейса.

6) Создание интерфейса, который мог бы быть реализован на различных платформах без значительных изменений в программном и аппаратном обеспечении.
7) Семантика интерфейса не должна зависеть от языка программирования.
8) Интерфейс должен быть потокобезопасным.

Стандарт включает в себя: коммуникационные операции «точка-точка», типы данных, коллективные операции, группы процессов, коммуникационный контекст, топологию процессов, средства управления средой разработки, информационные объекты, средства создания процессов и их управления, односторонние операции, расширенный интерфейс, средства параллельного ввода/вывода, языковые связки для *C* и *Fortran*, инструментарий поддержи.

В свою очередь, *MPI*-программу можно представить, как совокупность процессов, взаимодействующих между собой по средствам *MPI*-коммуникатора, операций обменов и примитивов синхронизации. В листинге 1.4 представлен пример *MPI*-программы «*Hello World*», написанной на языке C.

Листинг 1.4 -- *MPI*-программа «*Hello World*»

```
1    #include <stdio.h>
2    #include <mpi.h>
3
4    #define NELEMS(x) (sizeof(x) / sizeof((x)[0]))
5
6    int main(int argc, char *argv[])
7    {
8        int rank, commsize, len, tag = 1;
9        char host[MPI_MAX_PROCESSOR_NAME], msg[128];
10
11       MPI_Init(&argc, &argv);
12       MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13       MPI_Comm_size(MPI_COMM_WORLD, &commsize);
14       MPI_Get_processor_name(host, &len);
15
16       if (rank > 0) {
```

```
17          snprintf(msg, NELEMS(msg),
18                "Hello, master. I am %d of %d on %s",
19                rank, commsize, host);
20             MPI_Send(msg, NELEMS(msg), MPI_CHAR,
21                      0, tag, MPI_COMM_WORLD);
22         } else {
23          MPI_Status status;
24
25          printf("Hello, World. I am masted (%d of %d) on %s\n",
26                  rank, commsize, host);
27          for (int i = 1; i < commsize; i++) {
28              MPI_Recv(msg, NELEMS(msg), MPI_CHAR,
29                    MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
30              printf("Message from %d: '%s'\n",
31                      status.MPI_SOURCE, msg);
32          }
33         }
34         MPI_Finalize();
35         return 0;
36    }
```

Резюмируя, можно сказать, что *MPI* является неотъемлемой частью большемасштабных распределённых ВС и его программные реализации развиваются вместе с высокопроизводительными ВС.

1.6.2 User Level Failure Mitigation

В большемасштабных распределённых ВС высока вероятность того, что некоторая ошибка проявится как отказ процесса, что в итоге приведёт к прерыванию выполнения задачи, которая могла функционировать уже долго время. Однако, можно программно организовать процесс вычислений таким образом, чтобы при возникновении отказа, программа адаптировалась под новое количество параллельных ветвей и продолжала расчёты. Такая программа будет являться *отказоустойчивой* или, иначе говоря, *живучей*.

User Level Failure Mitigation (***ULFM***) – расширение стандарта *MPI*, предоставляющее средства, способствующие продолжению вычислений после возникновения отказа на этапе выполнения параллельной программы [10].

ULFM включает в себя ряд возвращаемых кодов и набор функций. Коды ошибок:

- *MPIX_ERR_PROC_FAILED* – отказ процесса прервал завершение операции *MPI*;
- *MPIX_ERR_PROC_FAILED_PENDING* – ошибка приёма сообщения от отправителя без определённого заранее идентификатора;
- *MPIX_ERR_REVOKED* – процесс вызвал операцию *MPI_Comm_revoke* на коммуникаторе.

Функции:

- *MPI_COMM_REVOKE*(*comm*) – прерывает любые операции на коммуникаторе *comm*;
- *MPI_COMM_SHRINK*(*comm*, *newcomm*) – создаёт новый коммуникатор *newcomm*, в котором нет отказавших процессов коммуникатора *comm*;
- *MPI_COMM_FAILURE_GET_ACK*(*comm*) – возвращает группу процессов, которые были определены как отказавшие к моменту последнего вызова функции *MPI_COMM_FAILURE_ACK*(*comm*);
- *MPI_COMM_AGREE*(*comm*, *flag*) – возвращает исключение о наличии отказа всем не отказавшим процессам.

С помощью вышеперечисленных средств, можно программно организовать отказоустойчивое выполнение *MPI*-программ.

2 Подходы к организации отказоустойчивости

Современные высокопроизводительные ВС являются большемасштабными. Тройку самых производительных ВС согласно списку *Top*-500 [9] составляют:

1) *Sunway TaihuLight* (10 649 600 процессорных ядер);
2) *Tianhe*-2 (*MilkyWay*-2) (3 120 000 процессорных ядер);
3) *Piz Daint* (367 760 процессорных ядер).

Оценки исследователей в области суперкомпьютеров показывают, что в таких ВС время безотказной работы варьируется от нескольких десятков минут до нескольких часов [3]. Несмотря на то, что в области аппаратного обеспечения активно разрабатываются новые способы уменьшения вероятности возникновения отказов компонентов ВС, всё же остаётся невозможным создание абсолютно надёжных устройств [7]. Исходя из вышенаписанного, становится важным вопрос программной организации отказоустойчивости.

2.1 Контрольные точки восстановления

Данный метод организации отказоустойчивости предполагает периодическое сохранение состояния системы в централизованное устройство хранения, что позволяет при возникновении отказа процесса вычислительного поля, продолжить расчёт с последней сохранённой согласованной глобальной контрольной точки (КТ) [2, 5, 23, 25, 31].

Создание КТ может быть выполнено на разных уровнях и иметь различный размер:

1) **Уровень операционной системы (ОС)**. Сохраняется содержимое пространства ядра и пользователя для всех процессов ОС. КТ включает в себя очень большой объём данных.
2) **Уровень ядра ОС**. За счёт внедрения дополнительных компонентов (модулей), сохраняется лишь необходимая информация: содержимое памяти конкретного процесса и состояние связанного с ним ядра. КТ имеет меньшей размер, чем при первом способе создания.

3) **Уровень системных библиотек**. Сохраняется содержимое памяти и состояние ядра с использованием средств, предоставляемых ОС для управления процессами. КТ имеет большой размер.
4) **Прикладной уровень**. Сохраняется минимально необходимый для восстановления объём данных. КТ имеет наименьший размер из всех вышеперечисленных способов её создания.

Чем выше уровень, на котором производится создание КТ, тем труднее процесс согласования локальных КТ и формирование из них глобальной КТ. Это связанно с возникающими в процессе выполнения параллельной программы особенностями.

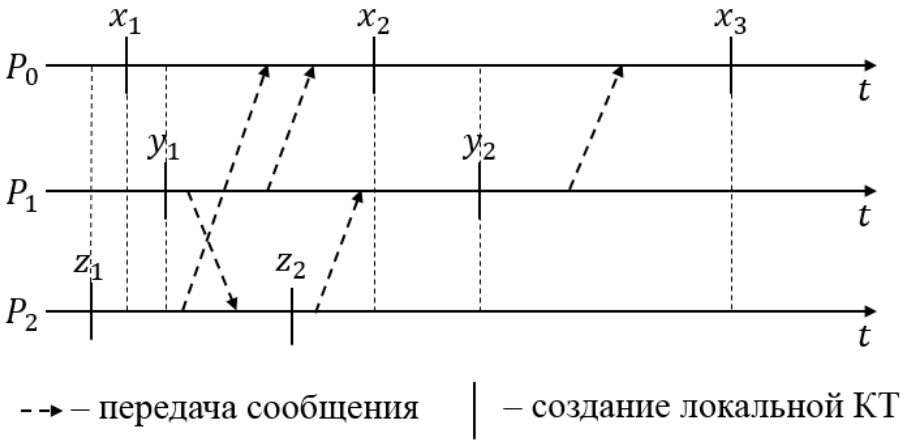

Рисунок 2.1 – Сообщения-сироты и эффект домино

На рисунке 2.1 представлена иллюстрация ситуации возникновения «*сообщений-сирот*» и «*эффекта домино*». Здесь показаны три процесса (P_0, P_1, P_2), взаимодействующие друг с другом через обмен сообщениями. Вертикальные черточки показывают на временной оси моменты создания локальной КТ. Штриховые стрелочки соответствуют процессу передачи сообщений и показывают моменты их отправления и получения. Если процесс P_0 сломается, то он может быть восстановлен с состояния x_3 без какого-либо воздействия на другие процессы.

Предположим, что процесс P_1 сломался после посылки сообщения и был возвращён в состояние y_2. В этом случае получение сообщения зафиксировано в x_3, а его посылка не отмечена в y_2. Такая ситуация не позволяет сформировать глобальную КТ из имеющихся локальных. Данное сообщение в таком случае

называется *сообщением-сиротой*. Процесс P_0 должен быть возвращён в предыдущее состояние x_2 и конфликт будет ликвидирован.

Предположим теперь, что процесс P_2 сломается и будет восстановлен в состояние z_2. Это приведёт к откату процесса P_1 в y_1, а затем и процессов P_0 и P_2 в начальные состояния x_1 и z_1. Этот эффект известен как *эффект домино*.

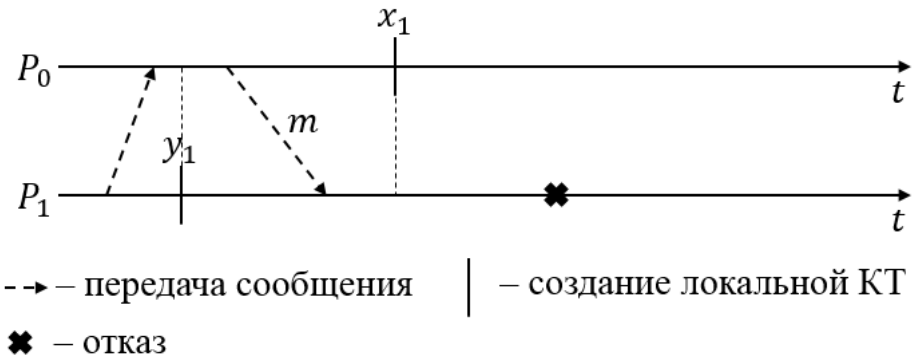

Рисунок 2.2 – Потеря сообщения

На рисунке 2.2 изображена ситуация *потери сообщения*: предположим, что локальные КТ x_1 и y_1 зафиксированы для восстановления процессов P_0 и P_1 соответственно, тогда, если процесс P_1 сломается после получения сообщения m, и оба процесса будут восстановлены, то сообщение m будет потеряно. Формирование глобальной КТ из имеющихся так же не представляется возможным.

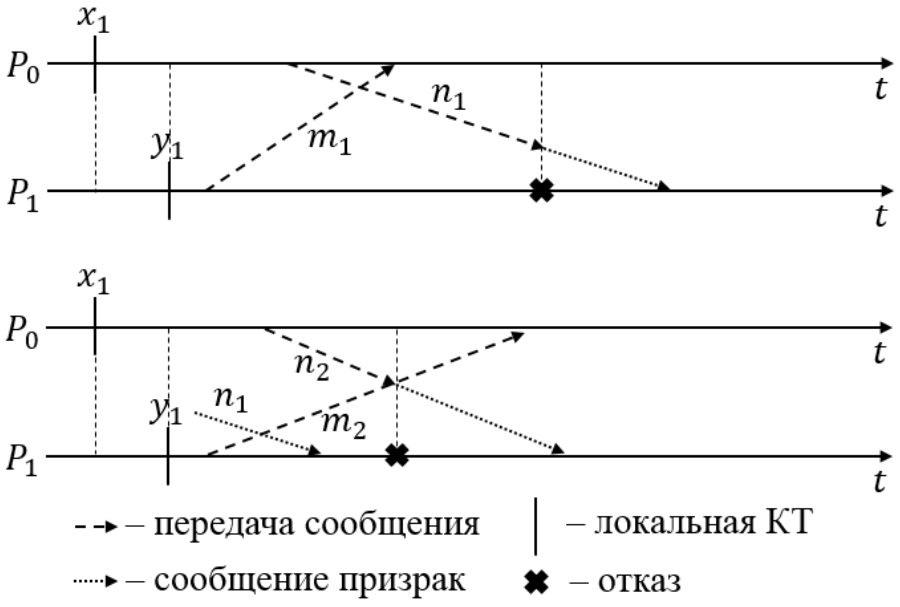

Рисунок 2.3 – Проблема бесконечного восстановления

На рисунке 2.3 проиллюстрирована проблема бесконечного восстановления: отказ происходит в процессе P_1 до получения сообщения n_1 от P_0. P_1 возвращается в состояние y_1, в котором нет записи о посылке сообщения m_1, поэтому P_0 должен вернуться в состояние x_1. После откатаP_1 посылает сообщение m_2 и принимает n_1 (*сообщение-призрак*). Процесс P_0 после отката в состояние x_1 посылает сообщение n_2 и принимает m_2. Однако после отката P_0 уже не имеет записи о посылке n_1, поэтому P_1 должен повторно откатиться к y_1. Теперь P_0 должен откатиться к x_1, поскольку он принял m_2, о посылке которого в y_1 уже нет записи. Это операция будет повторятся бесконечно.

Описанные выше трудности показывают, что глобальная контрольная точка, состоящая из произвольной совокупности локальных контрольных точек, не обеспечивает восстановления взаимодействующих процессов. Для распределённых систем запоминание согласованного глобального состояния является серьёзной теоретической проблемой.

Даже применение механизмов оптимизации времени создания и объёма КТ [17, 26-28] не даёт желаемый результат для систем Терафлопсного и Петафлопсного уровня. Некоторыми исследователями [1] отмечается, что в таких системах создание согласованной КТ занимает порядка 20-30 минут. Фактически, с учётом тенденции на увеличение количества компонентов ВС для наращивания производительности и уменьшением времени их безотказной работы, сама возможность успешного сохранения согласованной глобальной КТ оказывается под вопросом [1, 4].

В работах [12, 13] авторы предлагают разрабатывать новые принципы сохранения КТ за время, меньшее чем продолжительность безотказной работы ВС, и алгоритмы быстрого автоматического возобновления расчёта на исправной части рабочего поля. Данный подход сокращает время создания и объём КТ за счёт усложнения программного кода, так как предполагается непосредственное участие разработчика в организации отказоустойчивости.

Резюмируя, можно отметить, что современные высокопроизводительные ВС имеют большое количество компонентов, которые не являются абсолютно

надёжными, что в свою очередь уменьшает время безотказной работы ВС в целом. Применяя метод восстановления из КТ для организации отказоустойчивости, появляется ряд сложностей (длительное время создания КТ относительно продолжительности безотказной работы системы; объём КТ), которые ставят под вопрос данный подход. Один из вариантов преодоления этих сложностей заключается в делегировании обязанностей организации отказоустойчивости на разработчика, путём создания и применения новых принципов и алгоритмов формирования КТ на прикладном уровне и восстановления работы системы после отказа.

2.2 Живучие алгоритмы

Живучие ВС задействуют в любой момент времени все исправные ЭМ, входящие в её состав. Аналогично живучие алгоритмы подразумевают использование всех выделенных на решение конкретной задачи ресурсов ВС. То есть при отказе одного из процессов, алгоритм должен адаптироваться под новое количество параллельных ветвей. Следует отметить, что, когда речь идёт о живучести, важным является свойство децентрализованности – отсутствие главного процесса, отказ которого бы привёл к прекращению выполнения всего алгоритма.

Тривиального решения по преобразованию параллельного алгоритма в его живучий аналог нет, что позволяет разрабатывать оригинальные способы организации отказоустойчивости на теоретическом и прикладном уровнях.

Живучие алгоритмы в первую очередь должны:

1) Быть децентрализованными (все процессы равнозначны и взаимозаменяемы).
2) Адекватно реагировать на отказ одного или группы процессов (перераспределять вычисления по исправным процессам, восстанавливая расчётную область; производить перерасчёт потерянных значений).

3) Обладать основными свойствами классического алгоритма (дискретность, детерминированность, понятность, конечность, универсальность, результативность).

В данной работе рассматривается **живучий алгоритм решения СЛАУ методом Гаусса**. В листинге 2.1 представлен псевдокод прямого прохода, в листинге 2.2 – псевдокод обратного прохода, а в листинге 2.3 – код обработчика отказов на языке *C*. На рисунке 2.4 изображена блоксхема прямого прохода.

Листинг 2.1 – Псевдокод прямого прохода живучей версии алгоритма решения СЛАУ методом Гаусса

```
1    Function Gauss_Forward(matrix A, size n)
2        // Преобразование рабочей строки
3        For ROW = 0; ROW < n; ROW = ROW + 1
4            K = A[ROW][ROW]
5            For i = ROW + 1; i ≤ n; i = i + 1
6                A[ROW][i] = A[ROW][i] / K
7            A[ROW][ROW] = 1.00
8            // Если рабочая строка является последней в матрице A
9            If ROW = n - 1
10               Break
11           SIZE = (n – (ROW + 1)) / R
12           START = ROW + 1 + SIZE * RANK
13           // Если остались нераспределённые строки
14           If RANK = R – 1
15               SIZE = SIZE + (n – (ROW + 1)) mod R
16           // Обработка каждым процессом указанной части матрицы
17           For i = 0; i < SIZE; i = i + 1
18               K = 0.00 – A[START + i][ROW]
19               For j = ROW + 1; j ≤ n; j = j + 1
20                   A[START + i][j] = A[START + i][j] +
21                                     A[ROW][j] * K
22               A[START + i][ROW] = 0.00
23           // Генерация отказа (только один процесс за раз)
24           If RANK == 0
```

```
25              Failure_gen(chance)
26          MPI_Barrier(COMM)
27          i = R
28          MPI_Comm_size(COMM, R)
29          If R < i
30              ROW = ROW - 1
31              MPI_Comm_rank(COMM, RANK)
32              continue
33          // Обмен результатами вычислений
34          // между каждым процессом
35          SIZE = (n – (ROW + 1)) / R
36          For i = 0; i < R -1; i = i + 1
37              START = ROW + 1 + SIZE * i
38              Broadcast(A[START], SIZE, i)
39          // Рассылка последним процессом
40          SIZE = SIZE + (n – (ROW + 1)) mod R
41          START = ROW + 1 + SIZE * (R – 1)
42          Broadcast(A[START], SIZE, R - 1)
```

Из листинга 2.1 видно, что инструкции не отличаются от обычной, параллельной версии (листинг 1.2), за исключением строк 23-32, в которых с вероятностью *chance* генерируется отказ, для одного процесса, на каждом шаге прохода и, если отказ зафиксирован и обработан, происходит повтор данного шага с меньшим числом процессов.

Аналогично при обратном проходе – инструкции совпадают с теми, что указаны в листинге 1.3, за исключением строки 6, в которой с вероятностью *chance* генерируется отказ.

Листинг 2.2 – Псевдокод обратного прохода живучей версии алгоритма решения СЛАУ методом Гаусса

```
1   Function Gauss_Backward(matrix A, size n)
2       For i = n – 1; i ≥ 0; i = i – 1
3           X[i] = A[i][n]
4           For j = 1; j < n – i; j = j + 1
5               X[i] = X[i] – X[i-j] * A[n – j]
```

```
6            Failure_gen(chance)
7        Return X
```

Обработчик отказов, при возникновении соответствующей ситуации, о которой можно узнать благодаря коду возврата *MPI_ERR_PROC_FAILED*, выводит на экран информацию о том, какой процесс вышел из строя (строки 9-22). Далее с помощью функции *MPIX_Comm_agree* генерируется исключение о возникновении отказа для всех исправных процессов, после чего функция *MPIX_Comm_revoke* прерывает все операции на указанном коммуникаторе и по средствам *MPIX_Comm_shrink* создаётся новый коммуникатор, не содержащий неисправных процессов. Данный коммуникатор становится текущем, а старый обнуляется. Стоит заметить, что обработчик также перепривязывается.

Листинг 2.3 – Код обработчика отказов

```
1    void mpi_error_handler(MPI_Comm *comm, int *error_code, ...){
2        MPI_Group f_group;
3        int num_failures;
4        int loc_size;
5        char * ranks_failed = NULL;
6
7        MPI_Comm_size(*comm, &loc_size);
8        if( *error_code == MPI_ERR_PROC_FAILED ) {
9            printf("Process %d\n", mpi_mcw_rank);
10           /* Access the local list of failures */
11           MPIX_Comm_failure_ack(*comm);
12           MPIX_Comm_failure_get_acked(*comm, &f_group);
13           /* Get the number of failures */
14           MPI_Group_size(f_group, &num_failures);
15           ranks_failed = get_str_failed_procs(*comm, f_group);
16           printf("%2d of %2d) Error Handler: (Comm = %s) "
17                  "%3d Failed Ranks: %s\n",
18                  mpi_mcw_rank, mpi_mcw_size,
19                  (mpi_mcw_size == loc_size ? "MCW"
20                                            : "Subcomm"),
21                  num_failures, ranks_failed);
```

```
22              free(ranks_failed);
23
24              MPIX_Comm_agree(COMM, &loc_size);
25              MPIX_Comm_revoke(COMM);
26              MPIX_Comm_shrink(COMM, &NEWCOMM);
27              MPI_Comm_free(&COMM);
28              COMM = NEWCOMM;
29              MPI_Comm_set_errhandler(COMM, new_eh);
30          } else {
31              return;
32          }
33          /* Introduce a small delay to aid debugging */
34          fflush(NULL);
35          return;
36      }
```

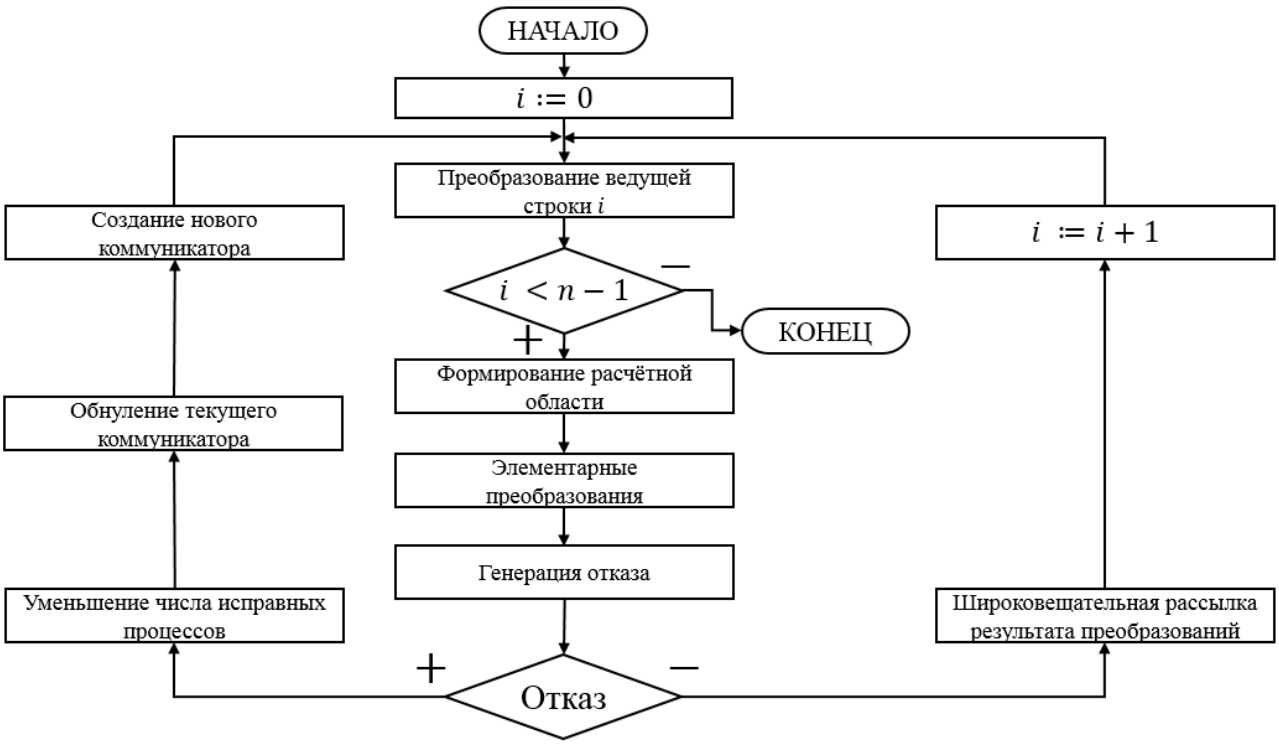

Рисунок 2.4 – Блоксхема прямого прохода живучей версии алгоритма решения СЛАУ методом Гаусса

Таким образом, рассмотренный алгоритм обладает высокими показателями живучести и способен продолжать работу даже при 1 исправном процессе.

3 Результаты экспериментов

Эффективность разработанного алгоритма исследовалась на кластере *Jet*. Необходимые характеристики данного кластера можно увидеть в таблицах 3.1 и 3.2. Кластер *Jet* укомплектован 18 вычислительными узлами, управляющим узлом, вычислительной и сервисной сетями связи, а также системой бесперебойного электропитания.

Таблица 3.1 – Конфигурация вычислительного узла кластера Jet

Процессор	Intel® Xeon® CPU E5420 (2.50 ГГц) x2
Количество ядер	8 x 2 = 16
Оперативная память	8 GB (4 x 2GB PC-5300)
Жесткий диск	SATAII 500GB (Seagate 500Gb Barracuda)

Таблица 3.2 – Конфигурация управляющего узла кластера Jet

Процессор	Intel® Xeon® CPU E5420 (2.50 ГГц) x2
Количество ядер	8 x 2 = 16
Оперативная память	16 GB (8 x 2GB PC-5300)
Жесткий диск	3 x SATAII 500GB (Seagate 500Gb Barracuda)

Рисунок 3.1 – График отказов (1024 уравнения)

Разработанная реализация живучей версии алгоритма решения СЛАУ методом Гаусса была запущена на кластере *Jet* на 8 узлах (по одному процессу на узел). Отказы генерировались на каждом шаге алгоритма с вероятностью 0.001 (равномерное распределение), с предположением, что в любой момент времени может произойти только один отказ. Размеры входной системы варьировались и принимали значения 1024, 2048 и 4096.

На рисунке 3.1 представлен график возникновения отказов в процессе выполнения алгоритма. Размер входной системы равен 1024 уравнения. Из графика видно, что произошло два отказа: на 854 шаге прямого прохода и на 697 шаге обратного.

Рисунок 3.2 – График отказов (2048 уравнения)

На рисунке 3.2 представлен аналогичный график возникновения отказов. Размер входной системы равен 2048 уравнения. Из графика видно, что произошло три отказа: на 1156 и 1823 шагах прямого прохода и на 1662 шаге обратного.

Из графика на рисунке 3.3 видно, что произошло четыре отказа: на 1666, 1901 и 3024 шагах прямого прохода и на 3750 шаге обратного.

Рисунок 3.3 – График отказов (4096 уравнения)

Рисунок 3.4 – График накладных расходов

Во всех случаях алгоритм был корректно завершён и его результат совпадал с результатами последовательной и параллельной версиями.

Для оценки накладных расходов на организацию отказоустойчивости и генерацию отказов был построен график зависимости времени выполнения параллельной и живучей версий алгоритма от количества уравнений во входной системе (рисунок 3.4). Из данного графика видно, что накладные расходы на организацию отказоустойчивости алгоритма решения СЛАУ методом Гаусса линейно зависят от интенсивности потока отказов и размера входной системы. Относительная незначительность накладных расходов (менее 10%) связанна с выбором способа организации параллельных вычислений. Так, например, при циклическом распределении строк в параллельной версии алгоритма (не живучей) накладные расходы значительно увеличатся, за счёт более эффективного использования ресурсов ВС параллельными процессами.

ЗАКЛЮЧЕНИЕ

В рамках данной выпускной квалификационной работы магистра были рассмотрены основные подходы к организации отказоустойчивости в большемасштабных вычислительных системах. Анализ показал неэффективность использования подходов, основанных на использовании контрольных точках восстановления, за счёт длительности создания согласованной глобальной контрольной точки относительно времени безотказной работы компонентов современных высокопроизводительных вычислительных систем.

Посредством языка программирования *C*, стандарта *MPI* и его расширения *ULFM* была разработана реализация живучей версии алгоритма решения системы линейных алгебраических уравнений методом Гаусса, эффективность которой подверглась экспериментальной оценки. В результате было установлено, что разработанная реализация имеет небольшие накладные расходы на организацию отказоустойчивости и генерацию отказов относительно предложенной параллельной версии алгоритма.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Cappello, F. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities / Cappello F. // International Journal of High Performance Computing Applications. — 2009. — Vol. 23, № 3. — P. 212–226.

[2] Elnozahy, E.N. A Survey of Rollback-Recovery Protocols in Message-Passing Systems /E.N. Elnozahy, L. Alvisi, Y. Wang, D.B. Johnson // ACM Computing Surveys. — 2002. — Vol.34, No. 3 — P. 375–408.

[3] Hsu, C.-H. A power-aware run-time system for high-performance computting / C.-H. Hsu, W.-C. Feng. // Proceedings of SC|05: The ACM/IEEE International Conference on High-Performance Computing, Networking, and Storage (Seattle, Washington USA November 12 – 18, 2005). — IEEE Press, 2005. — P. 1–9.

[4] Kogge, P.M. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems — Tech. Report TR-2008-13. — Univ. of Notre Dame, CSE Dept. — 2008. / P.M. Kogge, et al. URL: http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf

[5] Koren, I. Fault-Tolerant Systems / I. Koren, C. M. Krishna — San Francisco, CA: Morgan Kaufmann Publishers Inc., 2007. — 378 p.

[6] MPI 3.0 Documentation [Электронный ресурс]. – URL: http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[7] Sorin, D. Fault Tolerant Computer Architecture. Synthesis Lectures on Computer Architecture / D. Sorin — Morgan&Claypool, 2009. — 104 p.

[8] S. Torquato, Y. Jiao Dense packings of the Platonic and Archimedean solids // Nature. — 2009. — Т. 460, вып. 7257. — С. 876–879.

[9] Top-500 [Электронный ресурс]. – URL: https://www.top500.org/.

[10] ULFM [Электронный ресурс]. – URL: http://fault-tolerance.org/category/ulfm/

[11] Бовет. Д., Чезати М. Ядро Linux, 3-е изд.: Пер. с англ. // Спб. БХВ-Петербург, 2007. – 1104 с.: ил.

[12] Бондаренко А.А., Якобовский М.В. Обеспечение отказоустойчивости высокопроизводительных вычислений с помощью локальных контрольных точек // Вестник Южно-Уральского государственного университета. Серия «Вычислительная математика и информатика». 2014. Том 3. № 3 С. 20-36.

[13] Бондаренко А.А., Якобовский М.В. Моделирование отказов в высокопроизводительных вычислительных системах в рамках стандарта MPI и его расширения ULFM // Вестник Южно-Уральского государственного университета. Серия «Вычислительная математика и информатика». 2015. Том 4. № 3 С. 5-12.

[14] Гайдай А.В. Алгоритм оптимизации использования мьютексов по результатам предварительного профилирования // Материалы международной научной студенческой конференции (МНСК-2016), Новосибирск, 2016.

[15] Гайдай А.В. Адаптивный алгоритм операции захвата мьютекса. // Российская научно-техническая конференция «инновации и научно-техническое творчество молодёжи», Новосибирск, 2016.

[16] Гайдай А.В. Оптимизация синхронизации параллельных программ для вычислительных систем с общей памятью // Материалы Двенадцатой Международной Азиатской школы-семинара «Проблемы оптимизации сложных систем», Новосибирск, 2016.

[17] Данекина, А.А. Применение методов хеширования к задаче формирования инкрементных контрольных точек / А.А. Данекина, А.Ю. Поляков // Материалы Российской научно-технической конференции "Информатика и проблемы телекоммуникаций". – 2010. – Т. 1. – с. 155.

[18] Евреинов Э.В., Хорошевский В.Г. Однородные вычислительные системы. Новосибирск: Наука, 1978.

[19] Ильин В. А., Позняк Э. Г. Линейная алгебра: Учебник для вузов. — 6-е изд., стер. — М.: ФИЗМАТЛИТ, 2004. — 280 с.

[20] Курносов М.Г. Модели и алгоритмы вложения параллельных программ в распределенные вычислительные системы // Диссертация на соискание ученой степени доктора технических наук – Новосибирск, СибГУТИ, 2016.

[21] Курносов М.Г., Пазников А.А. Основы теории функционирования распределенных вычислительных систем (практикум). – Новосибирск: Автограф, 2015. - 52 с.

[22] Лав Р. Ядро Linux: описание процесса разработки, 3-е изд. – М.: Вильямс // 2015. – 496 с.

[23] Молдованова О.В. Исследование современных средств создания контрольных точек восстановления параллельных программ / О.В. Молдованова, А.Ю. Поляков // Тезисы докладов Восьмой Российской конференции с международным участием «Новые информационные технологии в исследовании сложных структур (ICAM 2010)». – Томск: НТЛ, 2010. – с. 22.

[24] Ортега Дж. Введение в параллельные и векторные методы решения /линейных систем. М.: Мир, 1991.

[25] Поляков А.Ю. О восстановлении программ из контрольной точки / А.Ю. Поляков // Вестник ЮУрГУ. Серия "Математическое моделирование и программирование". – 2010. – № 35(211), № 6. – С. 91 – 103.

[26] Поляков А.Ю. О подходах к оптимизации времени создания и объема контрольных точек восстановления / А.Ю. Поляков, А.А. Данекина // Тезисы докладов Восьмой Российской конференции с международным участием "Новые информационные технологии в исследовании сложных структур (ICAM 2010)". – Томск: НТЛ, 2010. – с. 23. – ISBN 978-5-89503-440-8

[27] Поляков А.Ю. Оптимизация времени создания и объёма контрольных точек восстановления параллельных программ / А.Ю. Поляков, А.А. Данекина // Вестник СибГУТИ. – Новосибирск: СибГУТИ, 2010. – № 2. – С. 87 – 100.

[28] Поляков А.Ю. Параллельный алгоритм формирования инкрементных контрольных точек / А.Ю. Поляков // Материалы Международной научно-технической конференции "Суперкомпьютерные технологии: разработка, программирование, применение (СКТ 2010)". Таганрог: ТТИ ЮФУ, 2010. Т.1. С. 290 294. – ISBN 978-5-8327-0383-1

[29] Руднев А.С. Алгоритмы локального поиска для задач двумерной упаковки // Диссертация на соискание ученой степени кандидата технических наук – Новосибирск, НГУ, 2010.

[30] Смирнов А.Д. Архитектура вычислительных систем. М.: Наука, 1990.

[31] Таненбаум, Э. Распределенные системы: принципы и парадигмы / Э. Таненбаум, М. Ван Стеен — Санкт-Петербург: Изд-во Питер, 2003. — 877 с.

[32] Философский словарь / Под ред. И. Т. Фролова. – 7-е изд., перераб. и доп. – М.: Республика, 2001. – 719 с.

[33] Хорошевский, В.Г. Архитектура вычислительных систем [Текст] / В.Г. Хорошевский. – М.: МГТУ им. Н. Э. Баумана, 2008. – 520 с.

[34] Хорошевский, В.Г. Распределенные вычислительные системы с программируемой структурой [Текст] / В.Г. Хорошевский // Вестник СибГУТИ. – 2010. – № 2. – С. 3-41.