Федеральное государственное автономное образовательное учреждение
высшего образования «Южно-Уральский государственный университет
(национальный исследовательский университет)»

На правах рукописи

ЕЖОВА Надежда Александровна

МОДЕЛЬ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ ДЛЯ ОЦЕНКИ МАСШТАБИРУЕМОСТИ ИТЕРАЦИОННЫХ АЛГОРИТМОВ НА КЛАСТЕРНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ

Специальность 05.13.11 – Математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени
кандидата физико-математических наук

Научный руководитель:
СОКОЛИНСКИЙ Леонид Борисович,
доктор физ.-мат. наук, профессор

Челябинск – 2019

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

По определению Джеффа Ротенберга (Jeff Rothenberg) моделирование в самом широком смысле – это экономически целесообразное использование чего-то искусственного вместо чего-то реального с целью исследования последнего. Моделирование позволяет нам использовать то, что проще, безопаснее или дешевле, чем реальность, для достижения определенного результата. Это дает нам возможность взаимодействовать с действительностью в упрощенном виде, преодолевая сложность, опасность и необратимость реальности [135]. Научное программирование сегодня является в высшей степени непростым процессом. Современные высокопроизводительные вычислительные платформы представляют собой технически сложные и дорогие системы, включающие в себя, как правило, две соединительные высокоскоростные сети, многоуровневую иерархическую память, многоядерные процессоры и мультиядерные ускорители. Индустрия высокопроизводительных вычислений предлагает сегодня широкий набор инструментов для исследования эффективности и профилирования программ при их выполнении на высокопроизводительных вычислительных системах. Преимуществом такого подходя является высокая точность и объективность получаемых характеристик и показателей эффективности программы по отношению к целевой аппаратной платформе. В качестве основных недостатков можно выделить следующие: 1) дорогостоящее процессорное время тратится не на получение эффективного результата, а на определение оптимальной конфигурации программы, которая этот результат должна выдать; 2) дефекты в проектировании программы, приводящие к неэффективному использованию целевой аппаратной платформы, выявляются в финальной фазе после ее полной реализации, что приводит к необходимости серьезной переработки исходных текстов вплоть до замены выбранного алгоритма; 3) зачастую исследование параллельной эффективности программы проводится на малом количестве

процессорных узлов, что не гарантирует приемлемую эффективность на «боевых» конфигурациях более высокого порядка. Использование формальных аналитических подходов для предсказания поведения программы на целевой платформе в значительной мере может устранить указанные недостатки натурного тестирования, сделав анализ алгоритмов и программ более быстрым, легким и дешевым. Предсказание производительности остается сложной нерешенной проблемой в области высокопроизводительных вычислений [132].

Актуальность темы исследования

Актуальность темы диссертационного исследования основывается на следующих основных факторах:

1) появление в ближайшие 2-3 года суперкомпьютеров с производительностью более одного экзафлопса (10^{18} операций с плавающей точкой в секунду);
2) доминирование кластерной архитектуры в области высокопроизводительных вычислений;
3) необходимость разработки и исследования новых классов сверхмасштабируемых параллельных алгоритмов, ориентированных на вычислительные системы экзафлопсного уровня производительности;
4) потребность в адекватных и простых в использовании моделей параллельных вычислений, способных на ранних этапах проектирования предсказывать границу масштабируемости для определенных классов алгоритмов, ориентированных на высокопроизводительные вычислительные системы с кластерной архитектурой.

Рассмотрим эти факторы более подробно.

Проекты построения экзамасштабных вычислительных систем сегодня реализуются в нескольких странах [48]. В Соединенных Штатах в соответствии с проектом ECP (Exascale Computing Project) [193], являющимся частью федеральной программы NSCI (National Strategic Computing Initiative) [192], в лабораториях Министерства энергетики США к 2023 году должно быть установлено не менее трех экзамасштабных суперкомпьютерных систем, которые на тесте HPL (High-Performance Linpack) [121] будут способны показать производительность более одного экзафлопса [94]. В Китае в 2018-2019 годах были созданы по крайней мере три компьютерные системы с различными архитектурами, демонстрирующие производительность в сотни петафлопс. Базируясь на этих результатах, Китай планирует создать не менее двух экзамасштабных суперкомпьютеров с производительностью более одного экзафлопса на тесте HPL [126]. Европейской Комиссией в 2018 году запущен масштабный суперкомпьютерный проект EuroHPC [87,187], в котором принимают участие 25 европейских стран. В соответствии с этим проектом в Европе к 2023 году должны быть запущены две экзамасштабные вычислительные системы, по крайней мере одна из которых должна базироваться на европейских технологиях. В Японии Министерством образования, культуры, спорта, науки и технологий в 2014 году на базе Центра вычислительных наук института Riken совместно с компанией Fujitsu запущен проект по созданию экзамасштабной вычислительной системы Post-K [149]. В отличие от американского и европейского проекта компьютер Post-K будет включать в себя только 48-ядерные центральные процессоры с векторизацией, разрабатываемые компанией Fujitsu на базе технологии ARM [43]. Предполагается, что в компьютере Post-K будет более 100 000 процессорных узлов с такими процессорами. Запуск системы намечен на середину 2020 года. В России также выполняются проекты по созданию суперкомпьютеров экзафлопсной производительности, среди которых можно отметить проект

«Торнадо» Группы компаний РСК [183,194] и проект «Эльбрус-16СВ», выполняемый совместно Институтом электронных управляющих машин имени И. С. Брука и компанией МЦСТ [180]. В результате выполнения этих проектов к 2023 году в России должны появиться технологии, обеспечивающие возможность создания вычислительных систем экзафлопсного уровня производительности. В соответствии с этим возникает необходимость разработки моделей параллельных вычислений для систем такого масштаба.

Еще одной важной тенденцией в области высокопроизводительной вычислительной техники является возрастающее преобладание кластерной вычислительной архитектуры [122,151]. В рейтинге TOP500 [188] самых мощных суперкомпьютеров мира (редакция от 18.06.2019)) семь из первых десяти мест, включая первое и второе, занимают кластеры. В целом суперкомпьютеры с кластерной архитектурой занимают более 90% списка TOP500. Это связано с тем, что кластеры показывают наилучшее соотношение производительность/стоимость. Таким образом наиболее важной является разработка моделей параллельных вычислений для кластерной архитектуры.

Многие параллельные численные алгоритмы, используемые до сих пор, были разработаны для многопроцессорных систем с общей памятью. Дизайн таких алгоритмов ориентирован на рост производительности за счет увеличения тактовой частоты процессоров. Подобные алгоритмы, как правило, невозможно адаптировать для эффективного использования экзамасштабных систем с кластерной архитектурой [9]. В соответствии с этим необходимо разрабатывать принципиально новые параллельные алгоритмы для достижения экзамасштабного уровня параллелизма на системах с миллионами процессорных ядер [133]. Одним из ключевых требований при проектировании сверхмасштабируемых вычислительных алгоритмов для экзамасштабных систем с распределенной памятью является минимизация межпроцессорных коммуникаций [37]. Граница масштабируемости параллельного

алгоритма при этом становится ключевой характеристикой. Поэтому актуальной является задача разработки простых и адекватных моделей параллельных вычислений, позволяющих уже на ранних этапах проектирования алгоритма предсказать верхнюю границу его масштабируемости.

К настоящему моменту разработаны многие десятки моделей параллельных вычислений и продолжают создаваться новые модели. Обзор различных моделей параллельных вычислений можно найти в работах [28,99,104,132,144,170]. По мере развития многопроцессорных систем происходило совершенствование существующих и создание новых моделей параллельных вычислений с целью учесть особенности современных параллельных архитектур. Адекватность моделей достигалась ценой повышения сложности их использования, что препятствовало широкому применению моделей параллельных вычислений в практике параллельного программирования. Для уменьшения сложности модели разработчики стали ограничивать класс параллельных архитектур, для которого эта модель применима. Так появились модели параллельных вычислений для общей, памяти, для распределенной памяти, для иерархической памяти, для графических ускорителей и др. Такой подход в некоторой мере позволил уменьшить сложность моделей, но достичь приемлемого компромисса между простотой, универсальностью и адекватностью не удалось. Дальнейшее упрощение моделей параллельных вычислений без ущерба для универсальности и адекватности возможно путем ограничения класса алгоритмов, к которому применима модель. Таким образом актуальной является задача разработки моделей параллельных вычислений для отдельных классов алгоритмов.

Одним из важных классов алгоритмов являются итерационные алгоритмы с высокой вычислительной сложностью [33,74,75,141]. Такие алгоритмы эффективно применяются для решения систем линейных и нелинейных уравнений [66,89], линейных и нелинейных неравенств

[35,59,92,123,147,166], декомпозиции многомерных матриц [2], эллиптических дифференциальных уравнений в частных производных [1], для решения задач математического программирования и оптимизации [1,34,184,185], выпуклой разрешимости [32,73,101,169] и многих других. В силу этого разработка модели параллельных вычислений, ориентированной на итерационные алгоритмы с высокой вычислительной сложностью, является значимой.

Степень разработанности темы

Фундаментом моделей параллельных вычислений явились последовательные модели вычислений. Наибольший вклад в их разработку внесли такие учены как А. Ахо, Дж. Хопкрофт, Дж. Ульман [7], Дж. Шефердсон [140], К. Ельгот, А. Робинсон [49], Дж. Хартманис [76], С. Кук [40]. В результате была создана универсальная модель RAM для последовательных вычислений. На ее основе усилиями ученых С. Фочьюна, Дж. Уилли [57], Л. Голдшлагера [64], Р. Ладнера, М. Фишера [97] была создана первая модель параллельных вычислений PRAM для многопроцессорных систем с общей памятью. В усовершенствование и развитие модели PRAM внесли существенный вклад следующие ученые: К. Мелхорн, Ю. Вишкин [112], П. Гиббонс, И. Матиас [62], А. Аггарвал, А. Чандра, М. Снир [5] и другие. С появлением многопроцессорных систем с распределенной памятью стало очевидно, что модель PRAM не может быть адекватно адаптирована для таких систем. Первой моделью, соединяющей параллельные алгоритмы и мультипроцессоры с распределенной памятью, явилась модель BSP, предложенная Л. Валиантом [161]. Модель BSP оказала большое влияние на дальнейшее развитие моделей параллельных вычислений. С развитием вычислительных систем с массовым параллелизмом потребовалось уточнение модели BSP. Одним из наиболее важных таких уточнений явилась модель LogP, предложенная Д. Куллером, Р. Карпом, Д. Паттерсоном и др. [42]. Коллективом ученых во

главе с К. Шаузером была предложена модель LogGP [8], расширяющая модель LogP путем моделирования передачи длинных сообщений.

Появление технологии параллельного программирования на основе MPI явилось стимулом для дальнейших расширений модели LogP. Наиболее важными расширениями явились модель PlogP, разработанная под руководством У. Гроппа [71], модель LogPQ, разработанная под руководством С. Хоригучи [156], и модель LoGPC, предложенная Дж. Кубятовичем и А. Агарвалом [95]. К. Камерон и Р. Ге разработали модель $\log_n$P [26], учитывающую затраты на работу программного обеспечения промежуточного слоя.

Дальнейшее развитие моделей параллельных вычислений было связано с появлением многоядерных процессоров, вычислительных грид-систем и гетерогенных кластеров с многоядерными графическими ускорителями. Л. Валиант разработал модель параллельных вычислений Multi-BSP [162], расширяющую модель BSP применительно к многоядерным процессорам. Модель HiHCoHP, разработанная Ф. Каппелло, П. Фраиньо, Б. Мансом и А. Розенбергом [31], и модель D-BSP, предложенная Г. Биларди, С. Фантоцци, А. Пьетракаприной и Г. Пуччи, ориентированы на многоуровневые иерархические сети из вычислительных кластеров. Ж. Боске и Л. Пастор опубликовали в [21] модель HLogGP, допускающую наличие в кластере вычислительных узлов, имеющих различающиеся процессоры, модули памяти и сетевые адаптеры. Группой ученых под руководством Дж. Жана разработана модель m$\log_n$P, являющаяся расширением модели $\log_n$P [158], ориентированным на вычислительные кластеры с многоядерными процессорами. А. Голдман с коллегами предложили расширение модели BSP для графических ускорителей, работающих под управлением CUDA [12].

В ближайшие два года ожидается появление вычислительных систем экзафлопсного уровня производительности. Этот фактор стимулировал

Т. Стерлинга, Д. Коглера, М. Андерсона и М. Бродовича разработать прототип модели параллельных вычислений SLOWER [150], ориентированной на экзамасштабные вычислительные системы.

Среди российских ученых большой вклад в развитие параллельных технологий и вычислительных моделей был сделан в работах В.В. Воеводина, Вл.В. Воеводина [172], В.П. Гергеля, Р.Г. Стронгина [173], С.А. Лупина, М.А. Посыпкина [182], О.В. Сухорослова [186], И.Н. Коньшина [181] и некоторых других.

Анализ состояния дел в области разработки моделей параллельных вычислений показывает, что невозможно разработать адекватную и простую в применении универсальную модель, соединяющую все многообразие современных многопроцессорных архитектур со всем многообразием современных параллельных алгоритмов. Появление экзамасштабных вычислительных систем представляет собой беспрецедентный вызов и стимулирует ученых к дальнейшим усилиям по разработке вычислительных моделей, соединяющих определенные классы многопроцессорных архитектур с определенными типами алгоритмов.

Цель и задачи исследования

Цель диссертационной работы состояла в разработке и исследовании новой модели параллельных вычислений для итерационных алгоритмов применительно к многопроцессорным системам с распределенной памятью, позволяющей предсказывать границу масштабируемости алгоритма на ранних стадиях его проектирования, и построении на ее основе параллельного каркаса для кластерных вычислительных систем экзафлопсного уровня производительности. Для достижения этой цели необходимо было решить следующие задачи.

1. Разработать модель параллельных вычислений.

2. Создать на языке C++ параллельный каркас, основанный на разработанной модели.
3. Выполнить проектирование и реализацию визуального конструктора программ на языке C++ в соответствии с разработанными моделью и параллельным каркасом.
4. Провести вычислительные эксперименты для верификации предложенной модели и разработанного программного обеспечения.

Научная новизна

Новизна работы заключается в том, что впервые разработана адекватная и простая в использовании модель параллельных вычислений для многопроцессорных систем с распределенной памятью, позволяющая предсказать границу масштабируемости для итерационных вычислительно сложных алгоритмов на ранней стадии их проектирования.

Теоретическая и практическая значимость работы

Теоретическая значимость работы состоит в том, что разработанная модель параллельных вычислений BSF включает в себя стоимостную метрику, позволяющую аналитически с достаточной точностью оценить границу масштабируемости вычислительно сложного итерационного алгоритма, ориентированного на экзамасштабные вычислительные системы. *Практическая значимость* работы состоит в разработке компилируемого программного шаблона и визуального конструктора программ, позволяющих для BSF-алгоритма быстро создать правильно работающую параллельную реализацию на языке C++ с использованием библиотек MPI и OpenMP.

Методология и методы исследования

Методологической основой диссертационного исследования являются теория множеств, формализм Бёрда-Миртенса, математический анализ. Для

описания алгоритмов использовались функции высшего порядка и операции над списками. При разработке параллельного каркаса и конструктора приложений применялись методы объектно-ориентированного проектирования и язык UML. Для организации параллельных вычислений использовались схема программирования SPMD, фреймворк «мастер-рабочие», библиотеки MPI и OpenMP.

Положения, выносимые на защиту

На защиту выносятся следующие новые научные результаты:

1. Разработана модель параллельных вычислений BSF, ориентированная на вычислительно сложные итерационные алгоритмы для кластерных вычислительных систем, позволяющая предсказать границу масштабируемости алгоритма на ранних стадиях его проектирования.
2. Разработан компилируемый BSF-каркас на языке C++ с использованием библиотек параллельного программирования MPI и OpenMP, инкапсулирующий все аспекты, связанные с параллелизмом, и позволяющий быстро создавать параллельные реализации алгоритмов, представленных в виде операций над списками.
3. Выполнены проектирование и реализация визуального конструктора программ BSF-Studio, позволяющего автоматизировать процесс создания BSF-программ на языке C++.
4. С использованием BSF-каркаса созданы BSF-программы для известных численных итерационных алгоритмов, на базе которых выполнена верификация BSF-модели.

Степень достоверности результатов

Утверждение, связанное с оценкой границы масштабируемости алгоритма, сформулировано в виде теоремы, снабженной строгим доказательством. Теоретические построения подтверждены имитационным моделированием на кластерной вычислительной системе. Верификация модели BSF и соответствующего параллельного каркаса были выполнены на известных итерационных численных алгоритмах из различных областей вычислительной математики и физики.

Апробация результатов исследования

Основные положения диссертационной работы, разработанные модели, методы, алгоритмы и результаты вычислительных экспериментов докладывались автором на следующих международных научных конференциях и семинарах:

1. 2018 Global Smart Industry Conference (GloSIC 2018). Chelyabinsk, November 13-15, 2018. URL: https://glosic.susu.ru.
2. XIII международная конференция «Параллельные вычислительные технологии» (ПаВТ'2019). Калининград, 2-4 апреля 2019 г. URL: http://agora.guru.ru/pavt2019.
3. XII международная конференция «Параллельные вычислительные технологии» (ПаВТ'2018). Ростов-на-Дону, 1-5 апреля 2018 г. URL: http://agora.guru.ru/pavt2019.
4. 3rd Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists (Ural-PDC 2017). Yekaterinburg, October 19, 2017. URL: https://ural-pdc.org/2017.

Публикации соискателя по теме диссертации

Основные результаты диссертации опубликованы в следующих научных работах.

Статьи в изданиях, рекомендованных ВАК

1. Ежова, Н.А. Исследование масштабируемости итерационных алгоритмов при суперкомпьютерном моделировании физических процессов / Н.А. Ежова, Л.Б. Соколинский // Вычислительные методы и программирование: новые вычислительные технологии. –2018. –Т. 19, № 4. –С. 416–430.
2. Ежова, Н.А. Модель параллельных вычислений для многопроцессорных систем с распределенной памятью / Н.А. Ежова, Л.Б. Соколинский // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. -2018. –Т. 7, № 2. –С. 32-49.
3. Ежова, Н.А. Модель параллельных вычислений BSF-MR / Н.А. Ежова, Л.Б. Соколинский // Системы управления и информационные технологии. –2019. № 3(77). –С. 15–21.
4. Ежова, Н.А. Обзор моделей параллельных вычислений / Н.А. Ежова, Л.Б. Соколинский // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. –2019. –Т. 8, № 3. –С. 58-91.
5. Ежова, Н.А. Программная поддержка модели BSF / Н.А. Ежова, Л.Б. Соколинский // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. –2019. –Т. 8, № 4.

Статьи в изданиях, индексируемых в SCOPUS и Web of Science

6. Ezhova, N.A. Scalability Evaluation of Iterative Algorithms Used for Supercomputer Simulation of Physical processes / N.A. Ezhova, L.B. Sokolinsky // Proceedings - 2018 Global Smart Industry Conference, GloSIC 2018. –IEEE, 2018. –10 p. DOI: 10.1109/GloSIC.2018.8570131.
7. Ezhova, N. Verification of BSF Parallel Computational Model / N. Ezhova // Proceedings of the 3rd Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists. CEUR Workshop Proceedings. -2017. -Vol. 1990. P. 30–39. URL: http://ceur-ws.org/Vol-1990/paper-04.pdf.

В работах 1–6 научному руководителю Л.Б. Соколинскому принадлежит постановка задачи, Н.А. Ежовой – все полученные результаты.

Структура и объем работы

Диссертация состоит из введения, четырех глав, заключения и библиографии. Объем диссертации составляет 137 страниц, объем библиографии – 195 наименований.

Содержание работы

В первой главе, «Модели параллельных вычислений», рассматривается понятие модели параллельных вычислений, дается классификация различных моделей и формулируются требования к модели параллельных вычислений. Особое внимание уделяется формализму Бёрда-Миртенса, являющемуся теоретическим базисом для построения параллельных каркасов. Дается обзор известных моделей параллельных вычислений и параллельных каркасов.

Во второй главе, «Модель параллельных вычислений BSF», описывается новая модель параллельных вычислений *BSF* (*Bulk Synchronous*

Farm) – *блочно-синхронная ферма*, ориентированная на многопроцессорные системы с кластерной архитектурой и предназначенная для разработки итерационных численных алгоритмов с высокой вычислительной сложностью. Приводятся базисные концепции, лежащие в основе модели BSF. Описываются архитектура BSF-компьютера и структура BSF-алгоритма. Вводится стоимостная метрика, формулируется и доказывается теорема о границе масштабируемости BSF-алгоритма.

В третьей главе, «Программная поддержка модели BSF», описываются структура параллельного BSF-каркаса на языке C++ и визуальный конструктор BSF-Studio, которые можно использовать для быстрого создания BSF-программ.

В четвертой главе, «Верификация модели BSF», представлены результаты по верификации модели BSF тремя различными способами. Первый способ заключается в имитационном моделировании работы вычислительно-сложного итерационного алгоритма на кластерной вычислительной системе с использованием библиотеки MPI. Второй способ заключается в проверке адекватности модели BSF на основе итерационного алгоритма Якоби для систем линейных алгебраических уравнений (СЛАУ). Третий способ состоит в исследовании адекватности модели BSF путем ее применения для решения гравитационной задачи.

В заключении в краткой форме излагаются итоги выполненного диссертационного исследования, представляются отличия диссертационной работы от ранее выполненных родственных работ других авторов, даются рекомендации по использованию полученных результатов и рассматриваются перспективы дальнейшего развития темы.

ГЛАВА 1. МОДЕЛИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

В данной главе рассматривается понятие модели параллельных вычислений, дается классификация различных моделей и формулируются требования к модели параллельных вычислений. Особое внимание уделяется формализму Бёрда-Миртенса, являющемуся теоретическим базисом для построения параллельных каркасов. Дается обзор известных моделей параллельных вычислений и параллельных каркасов.

1.1. Определение и классификация

Модель вычислений представляет собой упрощенное, абстрактное описание компьютера. *Модель параллельных вычислений* – это фреймворк (система правил и ограничений) для описания и анализа параллельных алгоритмов и программ. Термин «*параллельный*» подразумевает, что некоторые шаги алгоритма (операторы программы) специфицируются как независимые и могут выполняться параллельно. Проектировщики новых компьютеров, разработчики алгоритмов и программисты могут использовать модель вычислений для оценки своей работы, включая соответствие архитектуры компьютера различным приложениям, вычислительную сложность алгоритма и прогнозируемое быстродействие программы на различных вычислительных системах. Хорошая модель вычислений облегчает работу разработчиков компьютеров, алгоритмов и программ тем, что позволяет адекватно отобразить выбираемое решение на реальные компьютеры [170]. Подобную модель вычислений в литературе называют «*соединяющей моделью* (*bridging model*)» [161]. Универсальная соединяющая модель (см. рис. 1 а) может быть применима к любым алгоритмам и любым компьютерам. Такой универсальной соединяющей моделью вычислений для последовательных компьютеров и алгоритмов явились архитектура фон Неймана [28] и модель RAM (Random

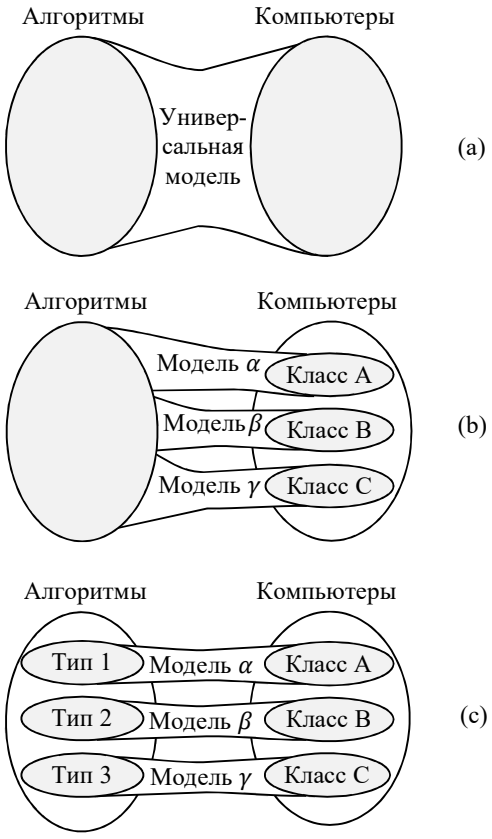

Рис. 1. Модели, соединяющие алгоритмы и компьютеры.

Access Machine) [7,40,49,76,140]. При появлении параллельных компьютеров были сделаны многочисленные попытки построить такую же универсальную соединяющую модель для параллельных алгоритмов [144], однако эти попытки потерпели неудачу. Это объясняется, главным образом, большим разнообразием многопроцессорных архитектур, появляющихся и развивающихся в настоящее время в погоне за повышением производительности.

В этих условиях создание простой, адекватной и универсальной модели параллельных вычислений является практически неразрешимой задачей. Для преодоления возникших трудностей был применен подход, схематично изображенный на рис. 1 b, в соответствии с которым многопроцессорные архитектуры были разделены на три класса: архитектуры с общей

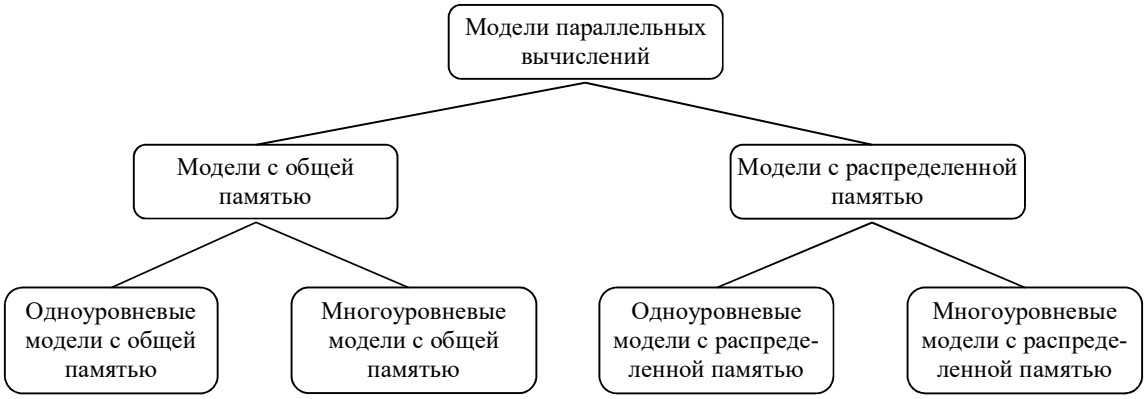

Рис. 2. Классификация моделей параллельных вычислений.

памятью, архитектуры с распределенной памятью и иерархические многопроцессорные архитектуры [170]. Для каждой из таких архитектур строились отдельные модели параллельных вычислений, которые, однако, оставались универсальными по отношению к множеству параллельных алгоритмов. Подобный подход позволил построить простые и универсальные модели с высоким уровнем абстракции, такие, как PRAM [57], BSP [161], LogP [42]. В целях адаптации таких моделей ко все более усложняющимся архитектурам многопроцессорных систем были предприняты многочисленные попытки их уточнения и расширения, что привело к появлению адекватных, но сложных для практического применения моделей параллельных вычислений (см., например, [15,102,127,130,131,168]). Исправить ситуацию в определенной мере позволяет подход, предполагающий разбиение всего множества алгоритмов по типам (см. рис. 1 с). Примерами различных типов алгоритмов могут быть итерационные численные алгоритмы, алгоритмы на графах, алгоритмы обработки больших данных и так далее. Для каждой пары (тип алгоритма, класс архитектуры) строится своя модель параллельных вычислений. Подобный подход позволяет достичь приемлемого компромисса между точностью оценок и простотой использования. В качестве примера

можно привести модель параллельных вычислений BSF [174,175], ориентированную на итерационные алгоритмы с высокой вычислительной сложностью и большие многопроцессорные системы кластерного типа.

В диссертации используется классификация моделей, изображенная на рис. 2. Все модели параллельных вычислений делятся на два класса: модели для многопроцессорных систем с общей памятью и модели для многопроцессорных систем с распределенной памятью. Каждый из этих классов, в свою очередь, делится на два подкласса: одноуровневые модели и многоуровневые модели.

Модели параллельных вычислений с общей памятью ориентированы на многопроцессорные архитектуры класса SMP и отчасти класса NUMA. Архитектура SMP (Symmetric MultiProcessor) – объединяет в себе симметричные мультипроцессоры, являющиеся наиболее распространенным подклассом многопроцессорных систем с общей памятью [36,191]. Отличительными особенностями SMP систем является то, что все процессоры, входящие в систему, являются одинаковыми, и каждый процессор имеет одно и то же время доступа к любой ячейке оперативной памяти. Наиболее популярными представителями класса SMP являются современные серверные системы, построенные на многоядерных процессорах [116]. Многопроцессорные системы с архитектурой NUMA (Non-Uniform Memory Access) [14] также имеют общую оперативную память, но в отличие от SMP время доступа процессора к различным ячейкам памяти может существенно различаться. В настоящее время многопроцессорные системы класса NUMA практически исчезли с рынка.

Модели параллельных вычислений с распределенной памятью ориентированы на массово-параллельные [145] и кластерные архитектуры [122]. В простейшем случае многопроцессорная система с распределенной памятью представляет собой совокупность однопроцессорных серверов, объединенных высокоскоростной соединительной сетью. Именно такую структуру

имели первые кластеры класса «беовульф» [190]. Однако, современные кластерные вычислительные системы имеют существенно более сложную структуру, включающую в себя SMP-узлы, внутри которых наряду с центральными многоядерными процессорами могут быть установлены дополнительные многоядерные ускорители, например, графические процессорные устройства (ГПУ) [117]. В этом случае может быть применено *двухуровневое* моделирование: сначала на уровне SMP-узла используется вычислительная модель для общей памяти, а затем на уровне кластерной системы в целом используется вычислительная модель для распределенной памяти.

Модели параллельных вычислений с многоуровневой (*иерархической*) *памятью* [134,191] ориентированы на современные SMP-системы, в которых кроме основной памяти имеются несколько уровней сверхоперативной кэш-памяти, встраиваемой в процессоры [22,77]. Сюда же можно отнести и новый вид быстрой энергонезависимой памяти Intel Optane на базе технологии 3D XPoint [164], занимающей промежуточное положение между оперативной памятью и накопителями класса SSD. В общем случае модель многопроцессорной архитектуры с многоуровневой (иерархической) памятью можно определить, как однородную систему, включающую в себя несколько уровней памяти с различным временем доступа. При этом определенные уровни памяти могут быть приватными для отдельных процессоров или их групп (в случае многоядерных сокетов под процессором понимается отдельное процессорное ядро). Модели параллельных вычислений для систем с многоуровневой памятью являются наиболее адекватными для приложений с интенсивным обменом данными между различными уровнями иерархической памяти. В качестве целевых аппаратных платформ для таких моделей часто фигурируют многоядерные кластеры с ГПУ, используемыми как ускорители, для которых характерно гибридное программирование с одновременным использованием технологий параллельных вычислений CUDA, OpenMP и MPI [165].

1.2. Требования к модели параллельных вычислении

Модель параллельных вычислений в общем случае должна включать в себя следующие пять компонентов, некоторые из которых в определенных случаях могут быть тривиальны [16]: *архитектурный компонент*, описываемый как помеченный граф, узлы которого соответствуют модулям с различной функциональностью, а дуги – межмодульным соединениям для передачи данных; *спецификационный компонент*, определяющий, что есть (синтаксически) корректный алгоритм/программа; *компонент выполнения*, задающий последовательность состояний архитектурных модулей, обеспечивающих корректное выполнение алгоритма/программы для определенных входных данных; *распараллеливающий компонент*, определяющий способы распределения вычислений, синхронизации и обмена данными между процессорными модулями, работающими независимо (параллельно); *стоимостный компонент*, определяющий одну или несколько метрик, позволяющих предсказать параметры выполнения алгоритма/программы (время выполнения, объем необходимой оперативной памяти и др.) в каждом конкретном случае.

В ранней работе Скилликорна [143] к модели параллельных вычислений предъявляются следующие требования: архитектурная независимость, согласованность, дескриптивная простота. *Архитектурная независимость* модели означает, что модель применима к широкому классу многопроцессорных архитектур. Программа, написанная согласно модели, не должна требовать переписывания при портировании на другую архитектуру, необходима только перекомпиляция. *Согласованность* предполагает, что время, предсказываемое стоимостными метриками модели для данной архитектуры, должно (асимптотически) соответствовать реальному времени выполнения программы. *Дескриптивная простота* предполагает, что модель абстрагируется от низкоуровневых механизмов параллельной обработки, межпроцессорных коммуникаций и синхронизации процессов, что обеспечивает легкость ее применения при практическом программировании.

В качестве наиболее важных свойств модели параллельных вычислений в современной литературе выделяют следующие [17,69].

– *Юзабилити*, определяющая легкость описания и анализа стоимости алгоритма средствами модели (модель должна быть легкой в использовании).
– *Переносимость*, характеризующая широту класса целевых платформ, для которых модель оказывается применимой.
– *Предиктивность*, выражающаяся в возможности с помощью стоимостных метрик модели предсказать *временные характеристики* выполнения алгоритмов/программ на целевой вычислительной системе.

Основными *временными характеристиками* являются следующие: *реальное* (*астрономическое*) *время* выполнения программы;

1) *абстрактное время* выполнения программы, позволяющее предсказать, какая из двух программ будет выполняться быстрее;
2) *масштабируемость* алгоритма/программы, определяющая максимальное количество вычислительных модулей в многопроцессорной системе, после которого прирост ускорения становится нулевым или отрицательным.

1.3. Обзор моделей параллельных вычислений

1.3.1. Одноуровневые модели с общей памятью

Одноуровневые модели с общей памятью являются наиболее ранними параллельными вычислительными моделями в историческом плане. Они ориентированы на симметричные мультипроцессорные системы, однако не учитывают иерархическую организацию памяти современных компьютеров.

***Модель параллельных вычислений PRAM* (*Parallel Random Access Machine*)** [85] предполагает, что все процессоры работают синхронно под

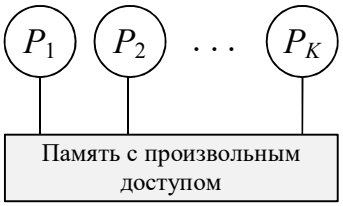

Рис. 3. *PRAM*-компьютер.

управлением одного тактового генератора и имеют произвольный доступ к общему адресному пространству оперативной памяти. Каждый процессор может выполнить арифметическую операцию, логическую операцию или операцию доступа к памяти за один машинный такт. Модель PRAM появилась в конце 1970-х годов как естественное расширение модели RAM (Random Access Machine) и была описана сразу в нескольких работах, среди которых можно выделить [57,64,97]. Модель PRAM получила широкое распространение и интенсивно использовалась при проектировании алгоритмов [84].

PRAM-компьютер (см. рис. 3) состоит из K процессоров, соединенных с общей памятью неограниченного размера. Каждый процессор имеет свой уникальный номер. Предполагается, что доступ к любой ячейке памяти происходит за один машинный такт. В общем случае каждый процессор может выполнять свою собственную программу, однако на практике большинство *PRAM*-алгоритмов реализуются по схеме SPMD (Single Program Multiple Data) [44,45], предполагающей, что все процессоры выполняют одну и ту же программу. На каждом машинном такте конкретный процессор может находиться либо в состоянии простоя, либо в активном состоянии. При этом все процессоры, находящиеся в активном состоянии, выполняют одну и ту же команду, но над разными данными.

Различают следующие три варианта модели PRAM в зависимости от политики разрешения конфликтов при параллельном доступе нескольких процессоров к одной и той же ячейке памяти в течении одного такта.

– *EREW* (*Exclusive Read Exclusive Write*) PRAM модель запрещает параллельный доступ к ячейке памяти и по чтению, и по записи.
– *CREW* (*Concurrent Read Exclusive Write*) PRAM модель разрешает параллельные чтения значения из ячейки, но запрещает параллельную запись в одну и ту же ячейку памяти.
– *CRCW* (*Concurrent Read Concurrent Write*) PRAM модель разрешает параллельный доступ к ячейке памяти как по чтению, так и по записи.

Указанные варианты модели PRAM характеризуются значительными различиями по времени выполнения параллельных алгоритмов. Рассмотрим следующую задачу: определить, являются ли все элементы двоичного массива $M[1\ldots n]$ нулями. Результатом вычислений в этом случае должно быть значение переменной A, являющееся результатом выполнения логической операции «ИЛИ» для всех элементов массива M. Параллельный CRCW PRAM алгоритм может решить эту задачу на n процессорах следующим образом. Первоначально переменной A присваивается значение 1. Затем все процессоры параллельно тестируют по одному элементу массива таким образом, чтобы индекс элемента совпадал с номером процессора. Если процессор обнаруживает в соответствующей ячейке значение 1, то он присваивает переменной A значение 0. Очевидно, что описанный CRCW PRAM алгоритм корректно решит поставленную задачу на n процессорах за $O(1)$ машинных тактов. В то же время, CREW PRAM алгоритм в общем случае потребует для решения этой задачи $\Omega(\log n)$ машинных тактов независимо от количества используемых процессоров [41]. С другой стороны, p-процессорная CREW PRAM модель может быть эмулирована с помощью p-процессорной EREW PRAM модели с замедлением не более, чем в $O(\log n)$ раз [88].

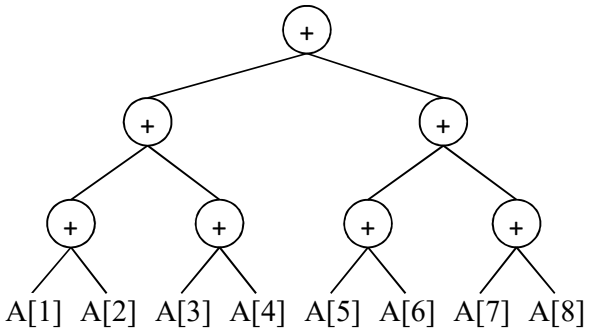

Рис. 4. Параллельное суммирующий элементов массива.

В теоретическом аспекте проектирование PRAM-алгоритма состоит в разработке стратегии, позволяющей достигнуть максимального уровня параллелизма, при котором задача решается за минимальное количество параллельных шагов, используя разумное количество процессоров. В рамках модели PRAM задача считается хорошо распараллеливаемой, если она может быть решена за $O((\log n)^m)$ параллельных шагов при некотором фиксированном m с использованием полиномиального количества процессоров. Данный класс задач получил название NC и был формально введен в работах [124,125] применительно к логическим схемам.

Для описания PRAM-алгоритмов удобно использовать *фреймворк WT* (*Work–Time*). В соответствии с WT-фреймворком PRAM-алгоритм представляется в виде последовательности шагов, каждый из которых представлен оператором одного из двух классов: последовательные операторы и параллельные операторы. Параллельные операторы специфицируются путем использования конструкций **pardo** и **for all**.

В качестве примера рассмотрим алгоритм PRAM SUM, суммирующий элементы числового массива $A[1..n]$, где $n = 2^l$ при некотором целом $l > 0$. Схема работы алгоритма PRAM SUM представлена на рис. 4. *PRAM*-алгоритм организуется как сбалансированное бинарное дерево, листьям которого соответствуют элементы массива A, а в качестве внутренних узлов

Алгоритм 1 PRAM SUM – параллельное суммирование

Input: Числовой массив $A[1..n]$ размера $n = 2^l$ в оперативной памяти.
Output: Сумма элементов массива A.

```
begin
   d = n
   for j = 1 to l
      d = d/2
      for all 1 ≤ i ≤ d pardo
         A[i] = A[2i-1] + A[2i]
      end for
   end for
   return A[1]
end
```

Рис. 5. Алгоритм PRAM SUM, суммирующий элементы массива.

фигурируют операции суммирования значений, представленных дочерними узлами. Алгоритм PRAM SUM выполняет вычисления снизу-вверх, осуществляя суммирование на каждом уровне за один параллельный шаг. При этом в контексте рис. 4, на первом уровне задействуется четыре процессора, на втором – два, и на третьем – один. WT-реализация алгоритма PRAM SUM приведена рис. 5. Из этой реализации видно, что решение состоит из k параллельных шагов. Учитывая, что $n = 2^l$, отсюда получаем $l = \log n$. Следовательно, вычислительная сложность алгоритма составляет $O(\log n)$ машинных тактов. При этом задействуется $n/2$ процессоров. Таким образом, задача суммирования элементов массива принадлежит классу NC и является хорошо распараллеливаемой в рамках модели PRAM. Очевидно, что границей масштабируемости алгоритма PRAM SUM является $K \le n/2$.

Отметим, что в общем случае временная сложность алгоритма в модели PRAM зависит от деталей его реализации, и в этом плане модель PRAM является низкоуровневой. Основным недостатком модели PRAM является то, что она не применима к многопроцессорным системам с распределенной памятью, так как не учитывает расходы на межпроцессорные коммуникации.

Модель PRAM позволяет строить чрезвычайно эффективные параллельные алгоритмы. Однако это достигается ценой следующих серьезных

ограничений: все процессоры работают синхронно и межпроцессорные коммуникации выполняются мгновенно. На практике во многих случаях модель PRAM оказывается неадекватной [146]. В соответствии с этим был предложен ряд расширений и улучшений модели PRAM, которые более соответствуют реальным параллельным архитектурам. Параллельные операции чтения и записи в память, допускаемые моделью PRAM, на реальных компьютерах могут приводить к конфликтам доступа к ячейкам памяти. Для предотвращения массовых конфликтов доступа к памяти в работе [112] была предложена ***модель модульно-параллельного компьютера*** **(*Module Parallel Computer*)**, в соответствии с которой общее адресное пространство делится на m модулей. Каждый модуль допускает только одно обращение в течении одного машинного такта. Указанная модель в определенной степени решает проблему конфликтов доступа к памяти, однако она игнорирует аспекты, связанные с пропускной способностью системной шины или соединительной сети. Другое, более реалистичное расширение модели PRAM, получившее название ***QRQW PRAM*** **(*Queue-Read Queue-Write PRAM*)**, было предложено в [63]. Для разрешения конфликтов доступа к памяти в модели QRQW PRAM используются очереди доступа по чтению и записи для каждой ячейки памяти. Время доступа к ячейке памяти прямо пропорционально длине соответствующей очереди. Указанный подход был обобщен в ***модели QSM*** **(*Queuing Shared Memory*)** [62], которая была разработана с целью показать, что модель параллельных вычислений с общей памятью может играть роль соединяющей модели для параллельных алгоритмов. Для этого была выполнена прямая и обратная эмуляция модели QSM средствами BSP и других релевантных моделей.

С развитием архитектур параллельных компьютеров становилось все более очевидным, что доступ к нелокальной памяти оказывает серьезное влияние на производительность. Этот факт попытались учесть авторы ***модели BPRAM*** (*Block Parallel Random Access Machine*) [5]. В этой модели в

качестве дополнительного параметра вводится латентность и предусматривается блочная передача сообщений между процессорами. При передаче блока данных между двумя процессорами на передачу первого машинного слова отводится l тактов, а на передачу остальных машинных слов, входящих в блок, отводится b тактов на каждое слово, причем b много меньше l. Модель ***PRAM(m)*** [105] учитывает пропускную способность системной шины путем ограничения количества ячеек памяти, одновременно доступных для параллельных обращений в память. Эта модель базируется на модели CRCW PRAM с тем отличием, что на каждом такте могут быть выполнены не более m обращений к памяти.

Стандартная модель PRAM предполагает, что параллельные процессы автоматически синхронизируются на каждом такте без дополнительных временных затрат. Это допущение не является реалистичным, так как на практике, с одной стороны, не требуется синхронизировать параллельные процессы на каждом машинном такте, и, с другой стороны, синхронизация процессов требует накладных расходов. Для учета первого фактора ***модели APRAM*** [39] и ***Asynchronous PRAM*** [61] допускают асинхронное выполнение процессов между двумя последовательными точками синхронизации. В отличие от этого ***модель XPRAM*** [160] предполагает периодическую глобальную синхронизацию асинхронных процессов. Однако, указанные модели не учитывают накладные расходы, связанные с синхронизацией процессов.

Модель YPRAM [96] базируется на следующих трех характеристиках многопроцессорного компьютера, состоящего из P процессоров: *латентность*, *пропускная неэффективность* и *рекурсивная декомпозиция*. Под *латентностью* $\delta(P)$ понимается максимальное время, необходимое для передачи одного машинного слова между любыми двумя процессорами или между процессором и общей памятью. *Пропускная неэффективность* опре-

деляется следующим образом. Пусть массив M из N чисел равномерно распределен между P процессорами. Пусть $\tau(N)$ – максимальное время, требуемое для перестановки элементов массива M среди всех возможных перестановок. Тогда пропускная неэффективность $\beta(P)$ вычисляется по формуле

$$\beta(P) = \lim_{N\to\infty} \frac{\tau(N)P}{N}. \quad (1)$$

Рекурсивная декомпозиция подразумевает, что многопроцессорный компьютер делится на два субкомпьютера с равным количеством процессоров и равным объемом разделяемой памяти, доступ к которой осуществляется на основе механизма EREW. Каждый субкомпьютер является самостоятельным компьютером и может быть рекурсивно разделен на две части. Для субкомпьютера из S процессоров время, затрачиваемое на M чтений из памяти, оценивается как

$$\Theta\left(\lambda(S) + m + \frac{M}{S}\beta(S)\right), \quad (2)$$

где m – максимальное число чтений одного процессора.

***Модель HPRAM* (*Hierarchical PRAM*)** [78] рассматривает многопроцессорную систему с общей памятью как динамически конфигурируемую иерархию синхронных *PRAM-подсистем*, работающих асинхронно по отношению друг к другу. Иерархия определяет политику синхронизации между независимыми PRAM-системами. Помимо инструкций, присущих классической модели PRAM, HPRAM включает в себя специальную команду *partition* (*разбиение*), добавляющую в модель управляемый асинхронный параллелизм. Команда partition логически разбивает PRAM-(под)систему из P процессоров на непересекающиеся подмножества и назначает каждому из них свой синхронный PRAM-алгоритм. Предусматриваются два варианта HPRAM-системы: *private HPRAM* и *shared HPRAM*. В *private HPRAM* команда partition разбивает память на непересекающиеся сегменты пропорционально размерам создаваемых PRAM-подсистем таким образом, что каждая

PRAM-подсистема имеет свое собственное адресное пространство, недоступное для других PRAM-подсистем. В отличие от этого в *shared HPRAM* весь объем памяти остается в равной мере доступным для каждой создаваемой PRAM-подсистемы. *HPRAM-алгоритм* представляет собой совокупность PRAM-алгоритмов, организованных в иерархию. Каждая точка в иерархии ассоциируется с соответствующей командой partition. Благодаря введению иерархического разделения общей памяти, модели YPRAM и HPRAM в определенной мере позволяют абстрагироваться от низкоуровневых механизмов параллельной обработки, межпроцессорных коммуникаций и синхронизации процессов, и в этом плане частично удовлетворяют требованиям Скилликорна к моделям параллельных вычислений [143], однако это достигается существенным усложнением стоимостных метрик.

В статье [54] предлагается ***модель PRAM-NUMA***, являющаяся расширением модели PRAM на многопроцессорные системы с распределенной памятью, работающие в режиме эмуляция общей памяти [55,128]. Подобные многопроцессорные архитектуры получили название NUMA (Non-Uniform Memory Access) [122]. В отличие от модели PRAM, модель PRAM-NUMA вводит новую метрику, задающую *дистанцию* (относительное расстояние) между процессорами. PRAM-NUMA-компьютер включает в себя следующие компоненты (см. рис. 6): T процессоров, разбитых на p групп по T_P процессоров; общедоступная глобальная разделяемая память; p модулей локальной памяти; соединительная сеть, способная определять дистанцию между процессорными группами.

Каждый процессор соединен с глобальной разделяемой памятью, и каждая процессорная группа соединена со своим собственным модулем локальной памяти. Соединительная сеть обеспечивает доступ процессоров одной группы к локальной памяти остальных групп, при этом латентность

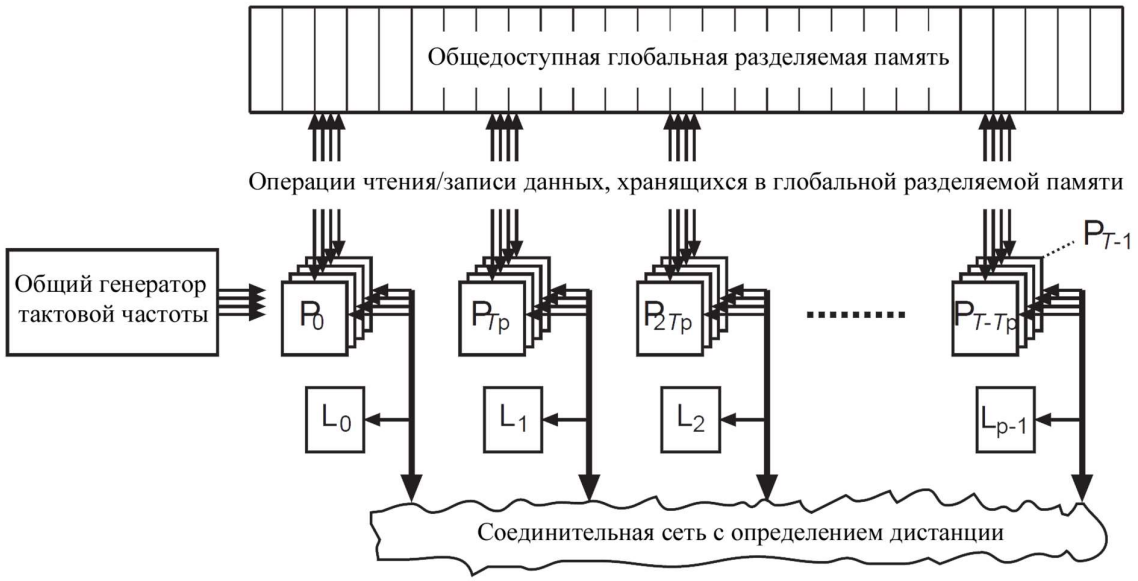

Рис. 6. PRAM-NUMA компьютер (P – процессор, L – локальная память).

маршрутизации прямо пропорциональна дистанции между процессором и модулем памяти, к которому он обращается. Пропускная способность каналов, соединяющих группу процессоров с общей и локальной памятью, является одинаковой. Каждый процессор может находиться либо в PRAM режиме, либо в NUMA режиме. В одной группе процессоров два или более процессора могут находиться в NUMA режиме. В этом случае они выполняют один поток команд над различными данными. Следует отметить, что NUMA-режим требует использования специального языка программирования «E» [53], сложного в использовании. В работе [56] предложено некоторое расширение модели PRAM-NUMA путем введения понятия «толстых» потоков управления, представляющих собой совокупность нитей управления, количество которых может меняться в ходе выполнения программы. Все нити одного «толстого» потока работают по схеме SIMD. Модель PRAM-NUMA ориентирована на многоядерные процессоры с нестандартной архитектурой и не может применяться для большинства современных многопроцессорных систем, поставляемых на рынке высокопроизводительной вычислительной техники.

1.3.2. Многоуровневые модели с общей памятью

Многоуровневые модели с общей памятью ориентированы на однородные многопроцессорные системы, в которых память делится на уровни с разным временем доступа [170]. Указанный класс моделей предназначен для более точной и реалистичной оценки времени доступа к памяти, организованной в виде иерархии уровней с монотонно возрастающими объемом памяти и временем доступа на каждом следующем уровне. Такие модели наиболее подходят для задач, связанных с перемещением больших объемов данных между различными уровнями иерархической памяти. В большинстве случаев модели этого класса вводят дополнительное множество измеряемых системных параметров, которые используются затем в качестве параметров стоимостных функций для повышения точности получаемых оценок.

Одной из первых последовательных моделей с многоуровневой памятью была ***модель HMM* (*Hierarchical Memory Model*)** [4]. HMM-компьютер содержит неограниченное количество регистров $R_1, R_2, R_3, \ldots$, каждый из которых может содержать целое число (или значение другого типа в зависимости от задачи). Операции выполняются так же как в модели RAM [7], за исключением того, что доступ к регистру R_i требует $\lceil \log i \rceil$ единиц времени, в то время как выполнение любой операции занимает одну единицу времени. Очевидно, что P-сложная задача, выполняемая на RAM-компьютере за время $T(n)$, может быть выполнена на HMM-компьютере за время $O(T(n)\log n)$.

***Модель BT* (*Hierarchical Memory with Block Transfer*)** [6] расширяет модель HMM путем введения понятия блока, представляющего собой совокупность ячеек памяти с последовательными адресами. Время доступа к ячейке памяти с адресом x определяется как $f(x)$. Однако, время доступа к блоку ячеек $[x-l, x]$ составляет $f(x)+l$. В качестве функции f могут фигурировать $f(x)=\alpha$, $f(x)=x^{\alpha}$ и $f(x)=\log x$, где α – некоторая константа. Арифметические операции, как и в модели RAM, занимают одну единицу

времени. В соответствии с этим сложение двух чисел, хранящихся в ячейках памяти с адресами x и y, и сохранение результата в ячейке памяти с адресом z занимает $f(x)+f(y)+1+f(z)$ единиц времени на BT-компьютере.

Еще одной ранней последовательной моделью вычислений для памяти с иерархическим доступом является ***модель LPM* (*Logarithmic Pipelined Model*)** [103]. Модель LPM предполагает, что доступ к m последовательным ячейкам памяти занимает $\log m$ единиц времени. Это предположение согласуется с технологией VLSI (Very Large Scale Integration) [111], предназначенной для создания проблемно-ориентированных сверхбольших микросхем, включающих в себя ЦПУ, ПЗУ и ОЗУ. Логические вентили, с помощью которых осуществляется доступ к ячейкам памяти в VLSI системах, организуются обычно в виде древовидной структуры. В соответствии с этим доступ к m последовательным ячейкам памяти не может быть осуществлен менее, чем за $\log m$ шагов.

Обобщением моделей HMM и BT является последовательная ***модель вычислений UMH* (*Uniform Memory Hierarchy*)** [11]. В рамках этой модели память представляется в виде иерархии модулей $\langle M_0, M_1, \ldots \rangle$. Например, M_0 может представлять регистровую память, M_1 – кэш-память, M_2 – оперативную память и так далее. Каждый модуль памяти M_u характеризуется тремя параметрами: $\langle s_u, n_u, l_u \rangle$. Параметр s_u обозначает количество ячеек памяти в одном блоке; n_u – количество блоков в модуле; l_u – количество тактов, необходимых для копирования одного блока из M_u в M_{u+1} или обратно. Модель UMH обобщает модели HMM и BT в том смысле, что в модели UMH полагается: $n_u = \alpha s_u$, $s_{u+1} = \rho s_u$, $l_u = \rho f(u)$ для всех $u = 0,1,\ldots$. Здесь α и ρ – положительные целочисленные константы, $f(u)$ – функция, определяющая количество тактов, необходимых для копирования одной ячейки памяти из

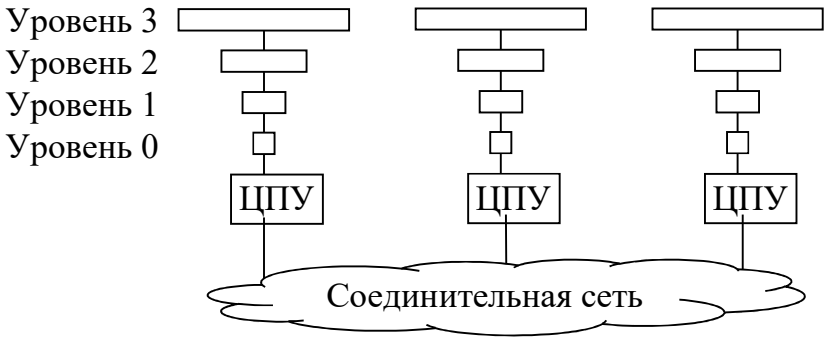

Рис. 7. Параллельная модель иерархической памяти.

уровня M_u в уровень M_{u+1}. В качестве функции f могут фигурировать, например, такие функции: $f(u)=1$, $f(u)=u$ или $f(u)=\rho^u$. В общем случае предполагается, что всегда имеет место неравенство $f(u)\le f(u+1)$.

Параллельные версии описанных многоуровневых моделей с общей памятью легко строятся путем репликации последовательной модели p раз. ***Модели P-HMM*** **и** ***P-BT*** [163] обобщают для параллельного случая модели HMM и BT соответственно. Параллельная модель с иерархической памятью строится как совокупность HMM или BT-компьютеров, объединенных соединительной сетью (см. рис. 7). При этом предполагается, что обмен данными по соединительной сети происходит через память уровня 0. Пропускная способность соединительной сети полагается такой, что сортировка массива, распределенного по модулям памяти уровня 0, выполняется за время $O(\log P)$, где P – количество процессоров в параллельной машине. Отметим, что в рамках современных представлений указанная модель является адекватной, если уровни $1,2,\ldots$ представляют внешнюю память.

Модель UPMH **(*Uniform Parallel Memory Hierarchy*)** [11] является параллельной версией модели UMH. UPMH-компьютер строится как совокупность UMH-машин, объединенных древовидной соединительной сетью. При этом для передачи сообщений используется память последнего уровня (самая медленная память).

1.3.3. Одноуровневые модели с распределенной памятью

Модели параллельных вычислений с распределенной памятью предполагают, что многопроцессорная вычислительная система строится как совокупность процессорных узлов, соединенных высокоскоростной коммуникационной сетью, обеспечивающей передачу сообщений от одного процессора другому. При этом каждый процессорный узел имеет свою приватную память, недоступную для других узлов. В этом разделе дается краткое описание моделей параллельных вычислений BSP, LogP, их модификаций, а также некоторых других одноуровневых моделей для многопроцессорных систем с распределенной памятью.

***Модель BSP* (*Bulk-Synchronous Parallelism*)** была предложена Валиантом (Valiant) в работе [161]. Данная модель широко используется при разработке и анализе параллельных алгоритмов и программ. BSP-*компьютер* представляет собой систему из K процессоров $P_1,\ldots,P_K$, имеющих приватную память $M_1,\ldots,M_K$, и соединенных сетью, позволяющей передавать данные от одного процессора другому (см. рис. 8). Для соединительной сети вводятся следующие характеристики: g – время, необходимое для передачи по сети одного машинного слова; L – время, необходимое для выполнения глобальной синхронизации. Мы будем предполагать, что время измеряется в машинных тактах. Моделирование передачи сообщений в BSP-компьютере реализуется с помощью понятия h-сессии. *h-сессия* является абстракцией произвольной коммуникационной операции, в ходе которой каждый процессор передает не более h машинных слов и получает не более h машинных слов. Время на выполнение одной h-сессии в BSP-компьютере не может превышать $h \cdot g$. *BSP-программа* состоит из n потоков команд, каждый из которых назначается отдельному процессору, и делится на *супершаги*, которые выполняются последовательно относительно друг друга. Каждый *супершаг*,

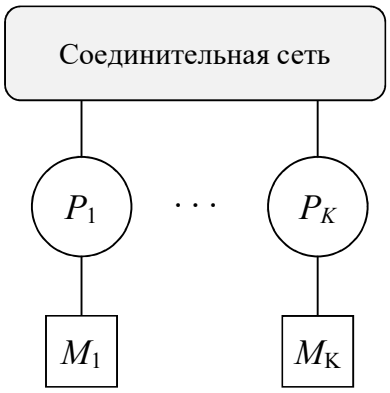

Рис. 8. BSP-компьютер.

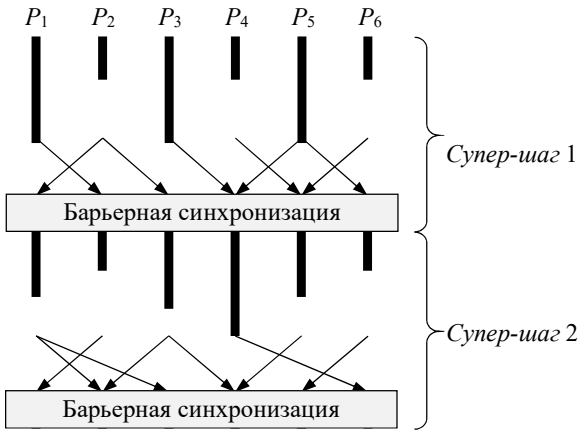

Рис. 9. Вычислительный процесс BSP.

в свою очередь, включает в себя следующие четыре последовательных шага: 1) вычисления на каждом процессоре с использованием только локальных данных; 2) глобальная барьерная синхронизация; 3) пересылка данных от любого процессора любым другим процессорам путем выполнения одной *h*-сессии; 4) глобальная барьерная синхронизация. Переданные данные становятся доступными для использования только после барьерной синхронизации. Пример BSP-программы из двух супершагов приведен на рис. 9. Жирными линиями обозначены локальные вычисления, тонкими линями со стрелками – пересылка данных.

Стоимостная функция в модели BSP строится следующим образом [68,154]. Пусть BSP-программа состоит из S супершагов. Предположим, что каждый процессор на i-том супершаге выполняет максимум w_i тактов в ходе локальных вычислений. Тогда общее время t_i, затрачиваемое системой на выполнение i-того супершага, вычисляется по формуле

$$t_i = w_i + h \cdot g + L. \tag{3}$$

Время T выполнения всей программы определяется по формуле

$$T = W + h \cdot g \cdot S + L \cdot S, \tag{4}$$

где $W = \sum_{i=1}^{S} w_i$. Значения W и S, как правило, зависят от количества процессоров K и от размера задачи.

При проектировании BSP-программы значения K, g и L рассматриваются как конфигурационные параметры многопроцессорной системы. Эффективный BSP-алгоритм должен минимизировать количество локальных вычислений, объем передаваемых данных и число глобальных синхронизаций при некоторых реалистичных значениях указанных параметров. Основными принципами разработки BSP-программ являются следующие.

– Балансировка загрузки процессоров, позволяющая сократить максимальное время локальных вычислений.
– Локальность данных, позволяющая сократить затраты на коммуникации.
– Крупнозернистый параллелизм, позволяющий сократить затраты на коммуникации и глобальную синхронизацию.

Балансировка загрузки процессоров касается и равномерного распределения вычислительной нагрузки, и пропорционального распределения данных между процессорами. Принцип *локальности данных* говорит о том, что в локальной памяти процессора необходимо хранить данные, к которым этот процессор обращается чаще всего. *Крупнозернистый параллелизм* подразумевает разделение программы на крупные параллельные секции, каждая из которых выполняется на отдельном процессоре, содержит значительный объем вычислений и не предполагает обменов данными в ходе этих вычислений. Значения параметров g и L для соединительной сети конкретного компьютера могут быть получены путем бенчмаркинга (эталонного тестирования). Методика бенчмаркинга в контексте модели BSP описана в работе [20]. Там же можно найти полученные значения параметров для некоторых реальных многопроцессорных систем.

В качестве примера рассмотрим простейший параллельный алгоритм, вычисляющий произведение квадратной матрицы A размера $n \times n$ на вектор b размера n. Для решения задачи задействуем $K = n + 1$ процессоров:

$P_0, P_1, \ldots, P_n$. Строки матрицы A распределим по процессорам следующим образом: *i*-тая строка a_i матрицы A будет храниться в памяти процессора P_i $(i = 1, \ldots, n)$. В память каждого процессора также поместим полный вектор b. Каждый процессор P_i параллельно выполняет умножение своей строки a_i на вектор b и получает одно число. Это число пересылается процессору P_0, который формирует результирующий вектор. Таким образом, мы имеем BSP-алгоритм, состоящий из одного супер-шага. Умножение строки a_i на вектор b состоит из n операций умножения и $n-1$ операций сложения. Следовательно, временные затраты на вычисления составят $w = O(n) + O(n-1) \approx O(n)$. Коммуникационные затраты составят gn в предположении, что $h = n$. Суммарные временные затраты, таким образом, составят $O(n) + gn + L$.

В работе [109] предлагается более сложный BSP-алгоритм умножения плотной матрицы на вектор, вычислительная сложность которого при $K = n$ составляет $O(n) + g\sqrt{n} + L$. Основным недостатком модели BSP является то, что она предполагает передачу сообщений длиной в одно машинное слово и не учитывает тот факт, что передача h машинных слов в виде одного сообщения может быть более эффективной, чем передача h сообщений длиной в одно машинное слово.

***Модель BSPRAM* (*Bulk Synchronous Parallel Random Access Machine*)**, предложенная в работах [110,153], является вариацией моделей BSP и PRAM. BSPRAM-компьютер состоит из K процессоров $P_1, \ldots, P_K$, каждый из которых имеет свою локальную память $M_1, \ldots, M_K$. В дополнение к этому имеется общая память, в равной мере доступная всем процессорам (см. рис. 10). Процессоры могут выполнять различные потоки команд. Вычислительный процесс в модели *BSPRAM* представляет собой

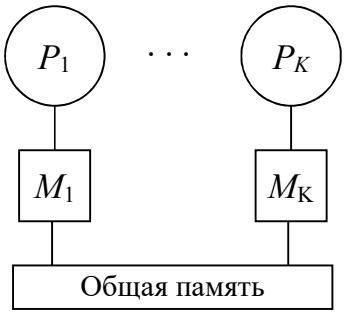

Рис. 10. *BSPRAM*-компьютер

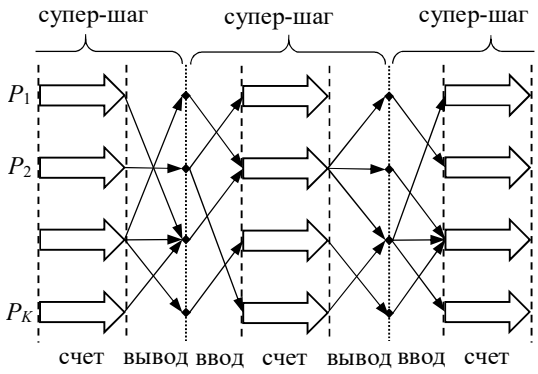

Рис. 11. Вычислительный процесс *BSPRAM*

последовательность супершагов (см. рис. 11). Супершаг включает в себя три фазы: *«ввод»*, *«счет»* и «*вывод»*. На фазе «ввод» процессоры читают данные из общей памяти; на фазе «вывод» процессоры пишут данные в общую память. Процессоры синхронизируются при переходе от текущего супершага к следующему; внутри супершага процессоры работают асинхронно. Указанный подход позволяет объединить преимущества моделей BSP и PRAM.

Модель параллельных вычислений LogP была предложена группой авторов в работе [42] как некоторое усовершенствование модели BSP. В качестве критики модели BSP указывались следующие факторы. На каждом супершаге количество данных, обрабатываемых одним процессором, должно быть примерно равным количеству слов, получаемых в ходе h-сессии, то есть h, что ограничивает зернистость параллелизма снизу. Далее, сообщения, передаваемые в конце выполнения супершага, не могут быть использованы реципиентом до начала следующего супершага. И последнее, модель BSP предполагает аппаратную поддержку механизма глобальной синхронизации, однако большинство многопроцессорных систем с распределенной памятью не имеют подобных аппаратных средств.

По аналогии с моделью BSP модель LogP предполагает, что компьютер состоит из процессорных модулей с приватной памятью, соединенных коммуникационной сетью. Основными стоимостными параметрами модели являются следующие.

L – (*latency*) верхняя граница латентности, представляющая собой время, необходимое для передачи сообщения длиной в одно машинное слово от одного процессорного модуля другому.

o – (*overhead*) накладные расходы, представляемые как промежуток времени, в течение которого процессорный модуль занят приемом или передачей сообщения; в это время процессорный модуль не может выполнять никакую другую работу.

g – (*gap*) задержка, представляющая собой минимальное время между двумя последовательными операциями чтения или передачи сообщений, выполняемыми процессорным модулем.

P – количество процессорных модулей.

При этом предполагается, что коммуникационная сеть имеет ограниченную пропускную способность, позволяющую каждому процессорному модулю получать или посылать одновременно не более $\lceil L / g \rceil$ сообщений. Если процессорный модуль пытается передать сообщение, перекрывающее этот лимит, то он переводится в состояние ожидания до тех пор, пока сообщение можно будет послать, не выходя за границу пропускной способности сети. Базовый вариант модели LogP предполагает, что все сообщения имеют небольшой размер (одно или небольшое количество машинных слов). Большие сообщения необходимо фрагментировать.

Время выполнения алгоритма в модели LogP определяется как максимум временных затрат среди всех процессорных модулей, участвующих в вычислениях. Время передачи одного короткого сообщения от одного процессорного модуля другому составляет $o + L + o$. Время доступа к элементу данных, располагающемуся в памяти другого процессорного модуля, будет равно $2L + 4o$. Для последовательной передачи n сообщений между процес-

сорными модулями P_1 и P_2 может быть организован конвейер, как это показано на рис. 12. В этом случае передача n последовательных сообщений займет время, равное $(n-1)g + o + L + o$ [91].

Сильной стороной модели LogP является ее простота. Однако, эта простота иногда может приводить к недостаточно точному предсказательному моделированию производительности алгоритмов и программ. Очевидным недостатком модели LogP является ограничение на размер сообщений. Попытка преодоления этого ограничения была сделана в ряде модификаций модели LogP, которые будут рассмотрены ниже.

В работе [8] было предложено расширение модели LogP путем добавления нового параметра G (*Gap per Byte*), задающего время, необходимое для передачи одного байта в составе длинного сообщения. Новая модель получила название ***LogGP***. В модели LogGP время передачи сообщения длиной в m байт вычисляется по следующей формуле:

$$T_n = o + (m-1)G + L + o. \qquad (5)$$

Более радикальное расширение модели LogP предложено в статье [90], где описывается ***параметризованная модель PlogP***, расширяющая модель LogP путем введения дополнительных параметров. Модель PlogP лучше учитывает особенности коммуникационного программного обеспечения такого, как MPI [71,72]. Среди этих особенностей выделяются следующие: 1) накладные расходы o и задержка g в значительной мере зависят от размеров сообщения; 2) накладные расходы, связанные с посылкой и приемом сообщения, могут существенно различаться, так как обработка асинхронно приходящих сообщений требует принципиально иной реализации, чем синхронные вызовы операции посылки. В соответствии с этим модель PLogP вместо параметров o и g вводит следующие параметры, зависящие от длины m сообщения:

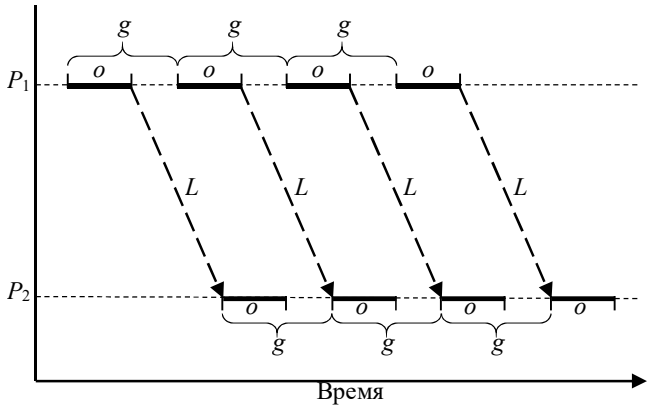

Рис. 12. Конвейерная передача сообщений в модели LogP.

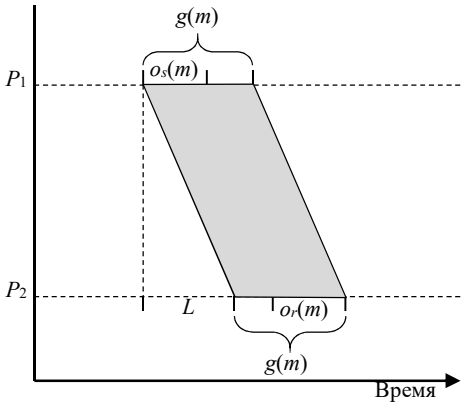

Рис. 13. Передача сообщения в модели PLogP.

$o_s(m)$ – накладные расходы, связанные с посылкой сообщения длиной n.

$o_r(m)$ – накладные расходы, связанные с приемом сообщения длиной n.

$g(m)$ – задержка между двумя последовательными операциями чтения или передачи сообщений длиной m.

Семантика параметра L (латентность) также несколько меняется. В модели PLogP латентность L обозначает время, затрачиваемое на передачу первого бита сообщения. Параметр P имеет тот же смысл, что и в модели LogP. Моделирование посылки сообщения длиной m между процессорными модулями P_1 и P_2 в метрике модели PLogP проиллюстрировано на рис. 13. Таким образом, время, затрачиваемое на передачу одного сообщения длины m, составляет $T_1 = L + g(m)$.

Соотношения между параметрами моделей PLogP и LogP/LogGP приведены в табл. 1. При этом в качестве короткого сообщения берется один байт. Время передачи одного короткого сообщения в LogP/LogGP составляет $o + L + o.$ В модели PLogP это время равно $L + g(1)$. Отсюда получается формула $o + L^{LogP/LogGP} + o = L^{PLogP} + g(1)$. Подставляя вместо o значение $(o_s(1) + o_r(1)) / 2$, получаем итоговое соотношение $L^{LogP/LogGP} = L^{PLogP} + g(1) - o_s(1) - o_r(1)$.

Табл. 1. Соотношения между LogP/LogGP и PLogP.

LogP/LogGP	**PLogP**
L	$L + g(1) - o_s(1) - o_r(1)$
o	$(o_s(1) + o_r(1)) / 2$
g	$g(1)$
G	$g(m) / m$ для $m \gg 1$
P	P

Табл. 2. Соответствие параметров LogPQ и LogP.

LogPQ	**LogP**
L	$L + o + (r-1)g$
o	$o + (r-1)g$
g	rg
P	P

В работе [156] была предложена ***модель параллельных вычислений LogPQ***, расширяющая модель LogP путем введения дополнительныхпараметров, связанных с обслуживанием очередей при передачи сообщений: SQ – размер очереди посылки сообщений, RQ – размер очереди приема сообщений и TQ – размер очереди передачи сообщений (см. рис. 14). Длина сообщений в модели LogPQ, также, как и в LogP, является постоянной и равна одному *коммуникационному* слову длиной w_c. Длинные сообщения представляются в виде последовательности сообщений длиной в одно *коммуникационное* слово. При этом длина w_p машинного слова может не совпадать с w_c . Соотношения между параметрами моделей LogPQ и LogP приведены в табл. 2, где $r = w_p / w_c$; L – латентность, представляющая собой время, необходимое для передачи сообщения длиной в одно коммуникационное слово; o – накладные расходы, представляемые как промежуток времени, в течение которого процессорный модуль занят приемом или передачей сообщения; g – временной интервал между двумя последовательными передачами сообщений от одного процессора другому.

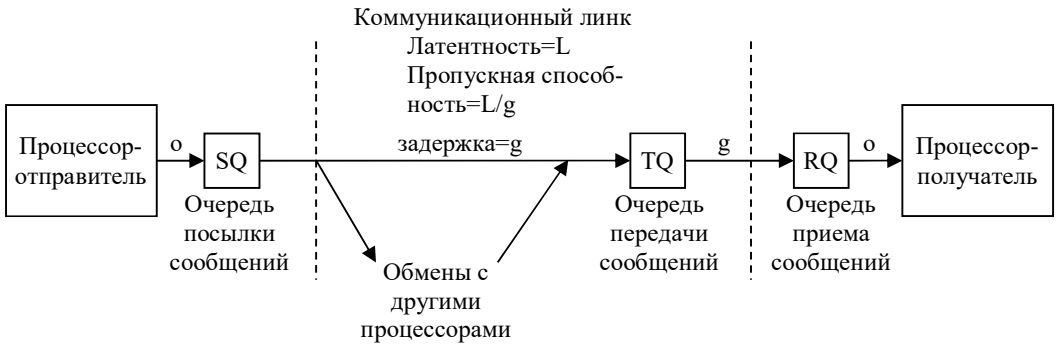

Рис. 14. Структура модели LogPQ.

Максимальная пропускная способность соединительной сети в модели LogPQ оценивается как $\lfloor L / g \rfloor + SQ + RQ$. Модель LogP, таким образом, может рассматриваться как частный случай модели LogP при $SQ = TQ = RQ = 1$. В работе [155] модель LogPQ была использована для оценки времени выполнения параллельного алгоритма перемножения матриц размера $n \times n$ на массивно-параллельном компьютере CM-5 [118]. Вычислительные эксперименты показали, что модель LogPQ более точно предсказывает реальное время выполнения программы по сравнению с LogP. Однако разница между двумя моделями оказалась существенной только для матриц небольшого размера ($n < 32$). Кроме этого, модель LogPQ оказалось сложно адаптировать к новым параллельным вычислительным архитектурам, и она не получила широкого распространения.

Модель параллельных вычислений LogGPS, представленная в работе [82], расширяет модель LogGP путем введения дополнительного параметра S, отражающего затраты на синхронизацию. Фактически модель LogGPS явилась адаптацией модели LogPQ к особенностям коммуникационных протоколов библиотеки MPICH [70], представляющей собой переносимую реализацию стандарта MPI. Еще одним расширением модели LogGP является ***модель параллельных вычислений LoGPC*** [113,114], которая учитывает накладные расходы, связанные с конфликтами, возникающими при обмене сообщениями в мультипроцессоре с распределенной памятью, а также

конвейерную передачу длинных сообщений с использованием интерфейса DMA [3,95]. Модель LoGPC является более точной по сравнению с LogGP, однако это достигается ценой значительного усложнения анализа алгоритмов. Кроме этого, LoGPC существенно зависит от аппаратных особенностей соединительной сети.

1.3.4. Многоуровневые модели с распределенной памятью

Модель параллельных вычислений log_nP [26,27] является расширением модели LogP. Модель $\log_n$P ориентирована на кластерные архитектуры, в которых передача сообщений между узлами с иерархической памятью осуществляется с помощью программного обеспечения промежуточного слоя такого, как MPI. Кроме явных затрат на передачу сообщений, учитываемых моделью LogP, модель $\log_n$P учитывает также неявные затраты, связанные с передачами данных между различными уровнями иерархической памяти, которые выполняются программным обеспечением промежуточного слоя при организации пересылок между процессорными узлами. Экспериментально было показано, что в некоторых приложениях при передаче сильно фрагментированных данных большого объема неявные затраты могут в несколько раз превосходить явные затраты на пересылку типа точка-точка. Стоимостная метрика модели $\log_n$P базируется на следующих пяти параметрах.

- l – *эффективная латентность* (*effective latency*), определяемая как разница между временем, затрачиваемым процессором на передачу фрагментированных и нефрагментированных данных. Эффективная латентность может быть вычислена с помощью платформенно-зависимой функции $f(s,d)=l$, где s – длина одного сообщения при передаче массива, d – длина пропусков между единичными сообщениями в передаваемом массиве.

o – *эффективные накладные расходы* (*effective overhead*) определяются как время, затрачиваемое процессором на передачу единичного сообщения в случае нефрагментированных данных ($d = 1$). Эффективные накладные расходы могут быть вычислены с помощью платформенно-зависимой функции $f(s,1) = o$.

g – (*gap*) задержка, представляющая собой минимальное время между двумя последовательными операциями передачи сообщений. В модели $\log_n$P предполагается, что $g = o$.

n – количество уровней иерархической памяти, используемых программным обеспечением промежуточного слоя при передаче сообщений. Фактически n определяет количество неявных обменов данными между различными уровнями иерархической памяти.

P – количество процессорных модулей.

Время, затрачиваемое на передачу одного сообщения в модели $\log_n$P определяется по формуле

$$T = \sum_{i=0}^{n-1}(o_i + l_i) = \sum_{i=0}^{n-1}(f_i(s,1) + f_i(s,d)). \quad (6)$$

Частным случаем модели $\log_n$P является модель $\log_3$P ($n = 3$), ориентированная на кластеры с SMP-узлами. В этом случае предыдущая формула преобразуется к виду

$$T = \sum_{i=0}^{2}(o_i + l_i).$$

Заметим, что при $n = 1$ модель $\log_n$P эквивалентна модели *Memory logP*, представленной в более ранней работе [25].

***Модель параллельных вычислений HiHCoHP* (*Hierarchical Hyper-Clusters of Heterogeneous Processors*)** [30,31] ориентирована на моделирование многоуровневых иерархических сетей из гетерогенных кластеров. На

пути от отправителя P_s к адресату P_r сообщение пересекает несколько уровней в иерархии сетей, упорядоченных по возрастанию отношения латентности к пропускной способности. Коммуникационная стоимость вычисляется как функция, зависящая от стоимостей $\sigma_s^{(k)}$ и $\sigma_r^{(k)}$ обработки сообщения на уровне k, стоимостей $\pi_s^{(k)}$ и $\pi_r^{(k)}$ упаковки и распаковки каждого пакета сообщения, и следующих трех параметров: 1) латентности $\lambda^{(k)}$, которая включает в себя латентности уровней ниже k; 2) пропускной способности $\beta^{(k)}$, включающей в себя пропускную способность сетей более низкого уровня; 3) ширины канала $\kappa^{(k)}$, вычисляемой как максимальное число пакетов, передаваемых за один такт по сети уровня k. Время передачи сообщения длиной m вычисляется по формуле

$$T(m) = \sigma_s^{(k)} + \pi_s^{(k)} p + \lambda^{(k)} + \frac{p-1}{\beta^{(k)}} + \sigma_r^{(k)} + \pi_r^{(k)} p,$$

где p – количество процессов. Модель HiHCoHP также включает параметр ρ_i, обозначающий процессорное время, затрачиваемое узлом P_i на выполнение единичного объема вычислений.

***Модель параллельных вычислений D-BSP* (*Decomposable Bulk Synchronous Parallel*)** [15] является расширением модели BSP. Модель D-BSP представляет многопроцессорную вычислительную систему как совокупность кластеров, каждый из которых имеет свою внутреннюю соединительную сеть, характеризующуюся определенной пропускной способностью и задержкой при передаче сообщений. Общее количество n процессоров в системе предполагается равным степени числа два. Для фиксированного i, такого, что $0 \le i \le \log_2 n$, n процессоров делятся на 2^i непересекающихся *i-кластеров* $C_0^{(i)},\ldots,C_{2^i-1}^{(i)}$ по $n/2^i$ процессоров в каждом. Процессоры каждого *i*-кластера могут обмениваться сообщениями независимо от остальных. Структура D-BSP машины формируется как бинарное дерево высоты $\log_2 n$, узлами которого являются *i*-кластеры. Указанное бинарное дерево обладает

следующими свойствами: кластер $C_j^{\log_2 n}$ содержит только один процессор P_j ($0 \le j < n$); $C_j^{(i)} = C_{2j}^{(i+1)} \cup C_{2j+1}^{(i+1)}$ для $0 \le i < \log_2 n$ и $0 \le j < 2^i$. Выполнение программы в модели D-BSP состоит из последовательности *мегашагов*. Каждый мегашаг представляет собой последовательность $1 + \log_2 n$ супершагов. В рамках *i*-того супершага процессоры выполняют некоторые вычисления и обмениваются данными исключительно в пределах их собственного *i*-кластера. В финале *i*-того супершага в каждом *i*-кластере выполняется барьерная синхронизация. Если на *i*-том супершаге каждый процессор выполняет не более w операций и передает или получает не более h сообщений, то время выполнения этого супершага оценивается по формуле $w + hg_i + l_i$, где g_i – задержка, а l_i – латентность для *i*-того уровня.

В работе [21] предлагается ***модель параллельных вычислений HLogGP* (*Heterogeneous LogGP*)**, являющаяся расширением модели LogGP и ориентированная на гетерогенные вычислительные кластеры. Модель HLogGP допускает наличие в кластере вычислительных узлов, имеющих различающиеся процессоры, модули памяти и сетевые адаптеры. Основная идея – заменить скалярные параметры на векторы и матрицы для учета специфики разнородных узлов кластерной вычислительной системы. Для моделирования гетерогенной вычислительной системы с M узлами вводится пять параметров: латентность L и байтовая задержка G представляют собой матрицы размера $M \times M$; величина накладных расходов o, задержка между сообщениями g и вычислительная мощность P являются векторами длины M. Если все эти параметры определены с достаточной точностью, модель HLogGP может достоверно предсказывать производительность параллельных алгоритмов, в которых коммуникационные затраты превалируют над вычислительными.

Модель параллельных вычислений LogfP, предложенная в [79], расширяет модель LogP на кластерные вычислительные системы, построенные

с использованием соединительной сети InfiniBand [86]. Модель LogfP предполагает, что первые f коротких сообщений передаются с нулевыми накладными расходами. Способность InfiniBand мгновенно пересылать сразу несколько коротких сообщений (в количестве, не превышающем f, после чего начинает наблюдаться замедление) обнуляет параметр g. Более того, использование в сети InfiniBand технологии RDMA (Remote Direct Memory Access) при передаче сообщений делает неадекватным параметр o_R, определяющий накладные расходы на стороне получателя. Параметры модели LogfP замеряются с помощью эталонного теста RTT (Round-Trip Time) по схеме $1:P$ - $P:1$. Модель LogfP позволяет повысить оценку производительности барьерной синхронизации на 40% по сравнению с другими известными моделями.

Модель параллельных вычислений PLP [115] ориентирована на иерархические гетерогенные вычислительные системы с однопроцессорными узлами и соединительной сетью Ethernet. Время передачи сообщения длиной m байт по схеме точка-точка вычисляется по формуле $T(m) = L + P_s + P_r$. Параметры P_s и P_r задают время, затрачиваемое процессорами отправителя и получателя на организацию передачи сообщения. Эти параметры в свою очередь зависят от физических параметров конкретных процессоров. Латентность L определяется следующим образом: $L = (n-1)G + L_t$, где G – межкадровая задержка при отправке n последовательных Ethernet-кадров, на которые разбивается сообщение, и L_t – латентность при кадровой передаче. Модель PLP применима как для обменов типа точка-точка, так и для коллективных коммуникаций.

Модель параллельных вычислений LogGOPS, предложенная в работе [80], представляет собой расширение модели LogGPS и предназначена для оценки масштабируемости параллельных алгоритмов, ориентированных на большие вычислительные системы, работающие под управлением MPI.

Согласно этой модели, стоимость передачи сообщения длиной m байт составляет $2o + L + (m-1)O + (m-1)G$. Параметры o, L, G имеют тот же смысл, что и в модели LogGPS. Параметр O является специфичным для модели LogGOPS и обозначает накладные расходы в пересчете на байт. Модель LogGOPS применяется для линейных коммуникаций типа scatter (один посылает всем) и gather (один получает от всех), для широковещательных коммуникаций на основе бинарного дерева и для распределенных коммуникаций, включающих в себя схему «всем от всех» и барьерную синхронизацию.

Модель параллельных вычислений LogGPH [168] является расширением модели LogGP и ориентирована на многопроцессорные системы, построенные на основе иерархии соединительных сетей. Для каждого уровня в сетевой иерархии вводится свой вектор параметров. Два процесса, запущенные на одном процессоре, взаимодействуют на уровне 1 и передают друг другу сообщения через общую память. Процессы, запущенные на разных процессорах одного узла, взаимодействуют на уровне 2. Процессы, запущенные на разных процессорных узлах, взаимодействуют на уровне 3, обмениваясь сообщениями через соединительную сеть. Время передачи сообщения длины m на сетевом уровне i вычисляется по формуле $T_i(m) = 2o_i + (m-1)G_i + L_i$.

В работе [162] предложена ***модель параллельных вычислений Multi-BSP***, расширяющая модель BSP в двух направлениях. Первое, Multi-BSP является иерархической моделью с произвольным количеством уровней, отражающих реальные технические особенности иерархической памяти и различных уровней кэш-памяти современных многопроцессорных систем. Второе, в качестве дополнительного параметра Multi-BSP включает в себя объем памяти на каждом уровне. Для каждого уровня i в иерархии вводится вектор параметров (p_i, g_i, L_i, m_i), где p_i – количество процессоров, g_i – задержка

при передаче сообщения, L_i – затраты на синхронизацию, m_i – объем памяти/кэша. Для системы, включающей в себя d уровней, общее количество процессоров вычисляется по формуле $P_d = \prod_{i=1}^{d} p_i$, суммарный объем памяти вычисляется по формуле $M_d = m_d + \sum_{i=1}^{d-1} m_i \prod_{j=i+1}^{d} p_j$, суммарная задержка вычисляется по формуле $G_d = \sum_{i=1}^{d} g_i$.

Модель mlog$_n$P [157,158] является расширением модели log$_n$P, ориентированным на вычислительные кластеры с многоядерными процессорами. Иерархию n-уровневой памяти, введенную в модели log$_n$P, авторы модели mlog$_n$P называют вертикальной. Дополнительно они вводят горизонтальную m-уровневую иерархию каналов передачи данных. По каналу нулевого уровня происходит обмен данными между ядрами одного процессора, канал первого уровня используется для обмена данными между ядрами разных процессоров одного процессорного узла, канал второго уровня – для обмена данными между ядрами разных процессорных узлов, и так далее. В соответствии с этим стоимость передачи сообщения на горизонтальном уровне i вычисляется по формуле

$$T_i = \sum_{j=0}^{n_i-1} (o_j^i + l_j^i) = \sum_{j=0}^{n_i-1} (f_j^i(s,1) + f_j^i(s,d)), \quad (7)$$

где семантика всех параметров с точностью до горизонтального уровня наследуется от модели log$_n$P.

Модель параллельных вычислений SLOWER, предложенная в работе [150], ориентирована на экзамасштабные вычислительные системы. Главной целью модели SLOWER является оценка производительности P вычислительной системы, которая вычисляется по формуле

$$P = e(L,O,W) \times s(S) \times \mu(E) \times a(R), \quad (8)$$

где e обозначает эффективность $(0 < e < 1)$, L – латентность, O – накладные расходы, W – задержка, вызванная ожиданием доступа к разделяемым ресурсам, s – степень параллелизма, S – мера недозагрузки, μ – скорость выполнения потока машинных команд, E потребление энергии, a – доступность, R – надежность. Формула (8) включает в себя критические с точки зрения модели SLOWER параметры и показывает их взаимное влияние на общую производительность системы. По мнению авторов модели SLOWER эффективность e и доступность a не оказывают значительного влияния на общую производительность системы. Мера доступности a отражает эксплуатационные проблемы, препятствующие способности системы выполнять реальную вычислительную работу. Она зависит от параметра R, определяющего среднее время между отказами системы. Пиковая скорость выполнения потока машинных команд μ зависит от тактовой частоты, которая в определенном диапазоне прямо пропорциональна энергопотреблению E. Степень параллелизма s определяется количеством процессорных ядер, одновременно выделяемых для решения задачи. Мера недозагрузки S вычисляется как отношение времени простоя процессорного ядра ко времени его работы при решении задачи. Недозагруженность возникает либо, когда процессорное ядро простаивает в ожидании работы или необходимых данных, либо в результате дисбаланса в распределении вычислительной работы между процессорными ядрами. Эффективность e вычислительной системы определяется как отношение скорости непрерывного выполнения машинных команд к пиковой скорости. Как следует из формулы (8), на эффективность влияет следующий ряд факторов. Латентность L представляет собой время, необходимое для инициализации доступа к удаленным данным или сервисам в незагруженной системе. Задержка W (ожидание отложенного доступа) возникает из-за конфликта доступа к разделяемым логическим или физическим ресурсам в загруженной системе. Накладные расходы O включают в себя

процессорное время, которое тратится на управление параллельными ресурсами и диспетчеризацию процессов. Модель SLOWER может применяться к широкому классу вычислительных систем экзафлопсного уровня производительности, однако в настоящее время отсутствует математическая формализация этой модели.

***Модель параллельных вычислений HLog$_n$GP* (*Heterogeneous Log$_n$GP*)**, предложенная в работе [102], развивает идеи, заложенные в моделях *LogGP* и *log$_n$P*. Модель HLog$_n$GP ориентирована на кластерные вычислительные системы с графическими ускорителями [117]. Модель HLognGP предполагает, что передача данных в ГПУ-кластере может происходить на трех различных уровнях: через оперативную память, через шину PCI и через соединительную сеть. В общем случае полагается, что количество таких *атомарных* коммуникационных уровней равно некоторой положительной константе n. В соответствии с этим стоимостная метрика модели HLog$_n$GP строится на основе следующих шести параметров:

- n – количество атомарных уровней при передаче сообщения от одного процессорного узла другому;
- L – латентность, представляющая собой вектор латентностей $(l_1,\ldots,l_n)$ для различных атомарных коммуникационных уровней;
- o – накладные расходы, представляющие собой вектор накладных расходов $(o_1,\ldots,o_n)$ для каждого атомарного коммуникационного уровня;
- g – (*gap*) задержка, представляющая собой вектор $(g_1,\ldots,g_n)$, определяющий для каждого атомарного коммуникационного уровня минимальное время между двумя последовательными операциями передачи сообщений;

G – (*gap per byte*) задержка на байт, представляющая собой вектор $(G_1,\ldots,G_n)$, определяющий для каждого атомарного коммуникационного уровня время, необходимое для передачи одного байта в составе длинного сообщения;

P – вычислительная производительность, представляющая собой пару (P_C,P_G), определяющую производительность ЦПУ и ГПУ соответственно.

Частный вид модели *HLog₃GP* имеет три атомарных коммуникационных уровня и соответствует ГПУ- кластерам. В этом случае время, необходимое для передачи одного сообщения между процессорными узлами, оценивается по формуле

$$T_{comm} = \sum_{k=1}^{3} L_k + 2\max(o_k, g_k) + (s-1)G_k, \qquad (9)$$

где s обозначает длину сообщения в байтах.

***Модель параллельных вычислений MBSP* (*Multi-memory BSP*)** [58] является расширением модели BSP и ориентирована на многопроцессорные системы с многоуровневой иерархической памятью и многоядерными процессорами. Стоимостная метрика модели MBSP базируется на семи параметрах (p,l,g,m,L,G,M), где параметры (p,l,g) наследуются от модели BSP, а (m,L,G,M) являются новыми параметрами, специфицирующими иерархическую память. Параметр M задает объем приватной «быстрой» памяти, имеющейся у каждого процессора. Параметр m задает объем «медленной» памяти, доступной всем ядрам процессорного узла. Стоимость доступа к медленной памяти определяется парой (L,G). Параметр G определяет время доступа к медленной памяти в пересчете на одно машинное слово. Параметр L задает латентность при обменах с медленной памятью. Время выполнения супершага вычисляется следующим образом: $\max(l,L)+w+gh+Gb$, где w – максимальное время вычислений, выполняемых одним процессором,

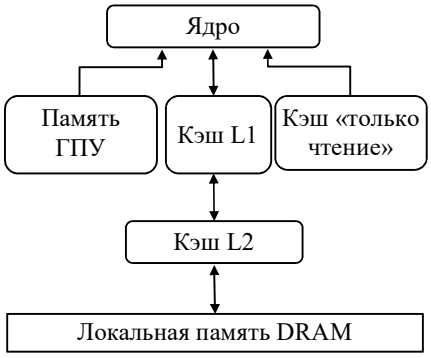

Рис. 15. Иерархия памяти в ГПУ с архитектурой Kepler.

h – максимальный размер сообщения, передаваемого или получаемого процессором, и b – максимальный объем обменов с медленной памятью, выполняемых одним процессором в ходе супершага.

В работе [12] предложено ***расширение модели BSP для ГПУ под управлением CUDA*** (далее для краткости – CUDABSP). Модель CUDABSP ориентирована на предсказание времени выполнения CUDA-программы на кластерных вычислительных системах с графическими ускорителями архитектуры Kepler (см. рис. 15) на основе оценок затрат на коммуникации и вычисления, которые выполняются независимо. Время, расходуемое на вычисления, оценивается на основе работы одного процессорного ядра и вычисляется по формуле

$$T_k = \frac{t \cdot (Comp + Comm_{DRAM} + Comm_{GPUM})}{R \cdot P \cdot \lambda}, \qquad (10)$$

где t – количество нитей, выполняемых на ядре, $Comp$ – время, расходуемое нитью на вычисления, $Comm_{DRAM}$ – затраты на обмены с локальной памятью DRAM процессорного узла, $Comm_{GPUM}$ – затраты на обмены с памятью ГПУ, R – тактовая частота, P – количество ядер в ГПУ, λ – положительное число, больше либо равное единицы, моделирующее эффект аппаратной оптимизации. Затраты на глобальные коммуникации оцениваются также, как в модели BSP.

1.4. Параллельные каркасы

Как указывалось в разделе 1.2, полноценная модель параллельных вычислений должна включать в себя спецификационный компонент определяющий, что есть корректная программа в контексте данной модели. Параллельная программа может выдавать правильный результат, однако быть некорректной по отношению к требованиям модели параллельных вычислений. Такого рода некорректность приводит к тому, что применение стоимостных метрик используемой модели будет выдавать неверные оценки. Для предотвращения таких ситуаций и для ускорения создания программных кодов на практике часто используют параллельные каркасы, разрабатываемые на основе требований соответствующей модели параллельных вычислений.

Параллельный каркас (*parallel skeleton*) в общем виде представляет собой программную конструкцию (или библиотеку функций), которая инкапсулирует некоторый шаблон параллельных вычислений и межпроцессорных коммуникаций [98]. В идеальном случае параллельный каркас представляет собой компилируемый, но неисполняемый код. Чтобы использовать параллельный каркас, программист должен добавить проблемно-зависимые типы и структуры данных, а также реализовать проблемно-зависимые функции. При этом дополнения программного кода могут делаться инкрементально с сохранением свойства компилируемости. Каркас берет на себя функцию объединения проблемно-зависимых операций с программным кодом, обеспечивающим их параллельное выполнение и коммуникации. Таким образом, абстрагирование от аспектов, связанных с параллелизмом, может значительно упростить и систематизировать разработку параллельных программ и помочь в эффективном использовании стоимостных метрик, преобразовании и оптимизации исходного кода. В силу высокого уровня абстракции параллельные каркасы имеют концептуальную общность с функциями высшего порядка, применяемыми в функциональном программировании, а

также с шаблонами объектно-ориентированного программирования, и многие конкретные каркасы используют эти механизмы.

В традиционных инструментах параллельного программирования используется подход, в соответствии с которым программист должен реализовывать низкоуровневое управление взаимодействием между параллельными процессами, используя механизмы, соответствующие используемой модели. Например, ядро MPI основано на обмене данными между процессами типа точка-точка с различными режимами синхронизации. Подобно этому, библиотеки потоковой обработки основаны на примитивах блокировки, условной синхронизации, атомарности и так далее. Этот подход является очень гибким, позволяя организовать взаимодействия любой сложности. Однако на практике многие параллельные алгоритмы основываются на хорошо понятных и хорошо известных шаблонах. Например, алгоритмы обработки изображений часто используют конвейерный параллелизм. Аналогичным образом, алгоритм для подбора параметра требует планирования параллельных процессов, которое не зависит от параметризованной задачи и диапазона значений исследуемого параметра.

Термин *параллельный каркас (algorithmic skeleton)* был введен Мюрреем Коулом (Murray Cole) в работе [38]. Появление этого термина явилось результатом осознания того факта, что многие параллельные приложения используют одни и те же внутренние коммуникационные шаблоны. Каркасный подход предполагает, что подобные шаблоны должны быть обобщены в виде программной конструкции или библиотеки функций, которые отвечают за реализацию скрытого управляющего «каркаса», оставляя программисту реализацию проблемно-зависимых функций, обеспечивающих решение конкретной задачи. Например, *конвейерный каркас* потребует от программиста реализацию операций, выполняемых на каждом шаге конвейера, в то время как каркас будет отвечать за распределение конвейерных операций между

различными процессорами, межпроцессорные коммуникации, непрерывность работы конвейера и так далее. Аналогичным образом, в *каркасе для подбора параметра* программист должен будет предоставить код для одиночного прогона параметризованной задачи и требуемый диапазон значений параметра. Каркас будет определять (и, возможно, динамически корректировать) количество используемых рабочих, гранулярность распределенных вычислительных заданий, коммуникационные механизмы и отказоустойчивость. На более высоком уровне абстракции использование *каркаса «разделяй-и-властвуй»* может потребовать от программиста описания способов разделения задачи на независимые подзадачи и последующего объединения результатов их работы, решения вопроса, является ли исходная задача делимой соответствующим образом, и реализации алгоритмов решения неделимых подзадач. Каркас в этом случае будет отвечать за все остальное, начиная с вопроса, стоит ли вообще использовать параллелизм, и кончая такими деталями, как динамическая диспетчеризация, гранулярность и коммуникации.

Подобный высокоуровневый подход меняет процесс разработки программы в нескольких отношениях. Во-первых, он освобождает пользователя от нетривиальной задачи принятия правильных проектных решений на основе многочисленных, взаимосвязанных низкоуровневых особенностей конкретного приложения и конкретной вычислительной платформы. Во-вторых, использование отлаженного и апробированного параллельного каркаса существенно уменьшает вероятность появления ошибок в результирующем коде, что особенно важно для сложных задач, решаемых на вычислительных системах с массовым параллелизмом, где традиционная отладка является слишком сложной. В-третьих, использование параллельного каркаса дает гарантию параллельной эффективности вместо апостериорной оценки производительности, в результате которой, возможно, придется отказаться от

Табл. 3. Таксономия базовых параллельных каркасов.

Тип каркаса	Область применения	Примеры элементарных каркасов
Параллелизм данных	Структуры данных	map, fork, reduce
Параллелизм задач	Задачи	farm, pipe

трудоемкой параллельной программы по причине ее фатальной неэффективности. В-четвертых, каркасы предоставляют семантически обоснованные методы для сборки и оптимизации программного кода, открывающие новые перспективы в разработке программного обеспечения (в том числе, в повторном использовании программных кодов). И последнее, но не менее важное: повышение уровня абстракции, заключающееся в переходе от частного к общему, дает новое понимание фундаментальных принципов параллельного программирования [98].

В зависимости от функциональности параллельные каркасы могут быть разделены на следующие три типа [67]:

– *каркасы с параллелизмом данных* (*data-parallel*), применяемые для выполнения одинаковых операций над элементами больших однородных структур данных;
– *каркасы с параллелизмом задач* (*task-parallel*), оперирующие задачами и обеспечивающие их синхронизацию в соответствии со связями по данным;
– *резолюционные* (*resolution*) *каркасы*, определяющие общий метод распараллеливания для семейства схожих задач.

Таксономия базовых параллельных каркасов приведена в табл. 3.

Map является наиболее важным базовым каркасом с параллелизмом данных. Его природа тесно связана с функциональными языками программирования. Семантика каркаса *map* заключается в том, что некоторая функция может быть одновременно применена ко всем элементам списка в целях

распараллеливания вычислений. Параллелизм данных организуется следующим образом: список разбивается на подсписки равной длины по числу процессорных узлов, и каждый процессорный узел обрабатывает свой подсписок параллельно с другими узлами. Обменов данными между процессорными узлами при этом не происходит. По завершению обработки полученные результаты объединяются в общий результат. Таким образом, каркас *map* соответствует схеме параллельной обработки SIMD (single-instruction multiple-data) [191].

Fork работает подобно *map*. Разница в том, что вместо применения одной и той же функции ко всем элементам списка, к каждому элементу применяется своя функция. Таким образом, каркас *fork* соответствует схеме параллельной обработки MIMD (multiple-instruction multiple-data) [191].

Reduce используется для вычисления инфиксных операций над списком путем деления элементов списка на пары и применения к каждой паре указанной инфиксной операции. Полученные результаты образуют список в два раза меньшей длины, к которому снова применяется *reduce*, и так да тех пор, пока не будет получено итоговое интегральное значение. Таким образом, в отличие от *map*, каркас *reduce* требует обменов данными между процессорными узлами для агрегирования частичных результатов.

Farm реализует параллельное выполнение задач по схеме мастер-рабочие. Этот каркас также называют *bag of tasks* (мешок задач). *Farm* обеспечивает параллельное выполнение независимых подзадач на узлах-рабочих и слияние полученных ими результатов в общий результат на узле-мастере, который финализирует решение задачи.

Pipe реализует конвейерный параллелизм. Этот каркас целесообразно использовать, когда одна и та же задача должна решаться для многих наборов входных данных. В этом случае решение задачи разбивается на последовательные стадии, количество которых называется длиной конвейера. Каж-

дая стадия выполняется отдельным процессорным узлом. Параллелизм достигается за счет того, что одновременно с выполнением *i*-той стадии для набора данных $x^{(j)}$ выполняется стадия $(i-1)$ для набора $x^{(j+1)}$. Каркас *pipe* обеспечивает синхронизацию выполнения стадий и передачу промежуточных результатов от одной стадии другой. Заметим, что в общем случае достаточно иметь каркас *pipe* с фиксированной длиной, так как каркасы могут вкладываться друг в друга.

1.5. Формализм Бёрда-Миртенса

Формализм Бёрда-Миртенса (*Bird–Meertens formalism*) или кратко *BMF* [18,19] является теоретическим фундаментом для проектирования и разработки параллельных каркасов. С точки зрения BMF параллельный каркас представляется как некоторый набор функций высшего порядка, оперирующих с данными, представленными в виде списков.

1.5.1. Основные понятия

Следуя работе [60] введем следующие обозначения. Применение функции $\mathbf{f}$ к элементу a обозначается как $\mathbf{f}.a$. Применение функции является ассоциативным справа, то есть $\mathbf{f}.\mathbf{g}.a = \mathbf{f}.(\mathbf{g}.a)$, и представляет собой инфиксный оператор с высшим приоритетом. Запись $a:\mathcal{A}$ означает, что элемент a имеет тип $\mathcal{A}$. Запись $\mathcal{A}\to\mathcal{B}$ обозначает множество всех функций, отображающих элементы множества $\mathcal{A}$ в элементы множества $\mathcal{B}$. Таким образом, если $a:\mathcal{A}$ и $\mathbf{f}:\mathcal{A}\to\mathcal{B}$, то $\mathbf{f}.a:\mathcal{B}$. Композиция двух функций обозначается с помощью инфиксного оператора $\circ$, то есть $(\mathbf{f}\circ\mathbf{g}).a = \mathbf{f}.\mathbf{g}.a$. Оператор $\circ$ имеет низший приоритет. Везде далее предполагается, что любая функция является всюду определенной. Тождественная функция обозначается как $\mathbf{id}$ и всегда возвращает то же самое значение, которое было указано в качестве аргумента: $\mathbf{id}.a = a$.

Бинарный оператор $\oplus$ определяется как функция $\oplus : \mathcal{A} \times \mathcal{B} \to \mathcal{C}$. Это означает, что оператор $\oplus$ любой паре аргументов $a : \mathcal{A}$ и $b : \mathcal{B}$ сопоставляет элемент $c : \mathcal{C}$. Это обозначается следующим образом: $a \oplus b = c$. При этом левая часть $(a \oplus)$ может рассматриваться как функция, имеющая тип $\mathcal{B} \to \mathcal{C}$. При фиксированном $a \in \mathcal{A}$ функция $(a \oplus)$ для произвольного аргумента b, возвращает значение $a \oplus \mathrm{b}$: $(a \oplus).b = a \oplus b$. Аналогичным образом правая часть $(\oplus \mathrm{b})$ может рассматриваться как функция, имеющая тип $\mathcal{A} \to \mathcal{C}$. При фиксированном $b : \mathcal{B}$ функция $(\oplus b)$ для произвольного аргумента a, возвращает значение $a \oplus \mathrm{b}$: $(\oplus \mathrm{b}).a = a \oplus b$. Таким образом, имеет место следующее свойство:

$$(a \oplus).b = a \oplus b = (\oplus \mathrm{b}).a. \qquad (11)$$

Указанная техника называется *секционированием* бинарного оператора и может эффективно применяться при эквивалентных преобразованиях выражений. Проиллюстрируем секционирование на арифметическом операторе сложения + на множестве вещественных чисел $\mathbb{R}$. Имеем $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Зафиксируем первый операнд операции +, пусть, например, это будет число 2. Тогда мы можем рассматривать функцию $(2+) : \mathbb{R} \to \mathbb{R}$, значения которой вычисляются по формуле $(2+).b = 2 + b$ для любого $b : \mathbb{R}$. Например, $(2+).3 = 2 + 3 = 5$.

Для бинарного оператора $\oplus : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ может существовать *нейтральный элемент* $\mathbf{id}_{\oplus} : \mathcal{A}$, обладающий следующими свойством:

$$a \oplus \mathbf{id}_{\oplus} = \mathbf{id}_{\oplus} \oplus a = a$$

для всех $a \in \mathcal{A}$.

Списком называется конечная последовательностей элементов одного и того же типа. Списки обозначаются квадратными скобками. Например,

$[1,\ 2,\ 3]$ – список из трех чисел, а $[[1],\ [1,2],\ [1,\ 2,\ 3]]$ – список из трех списков чисел. Тип $[\mathcal{A}]$ обозначает всевозможные списки из элементов типа $\mathcal{A}$, включая пустой список. Тип $[\mathcal{A}]_+$ обозначает всевозможные списки из элементов типа $\mathcal{A}$, исключая пустой список. Таким образом, если $n \geq 1$ и $a_1,...,a_n : \mathcal{A}$, то $[a_1,\ldots,a_n] : [\mathcal{A}]_+$. Функция преобразования элемента $a : \mathcal{A}$ в элемент типа $[\mathcal{A}]$ обозначается с помощью символа $[\cdot]$: $[\cdot].a = [a]$. Базовой операцией над списками является бинарная *операция конкатенации*, обозначаемая как $++$. Например,

$$[1] ++ [2] ++ [3] \ = \ [1,\ 2,\ 3].$$

Операция конкатенации является ассоциативной:

$$X ++ (Y ++ Z) = (X ++ Y) ++ Z$$

для любых $X, Y, Z : [\mathcal{A}]$. *Пустой список* обозначается пустыми квадратными скобками $[\ ]$ и является нейтральным элементом по отношению операции конкатенации:

$$X ++ [\ ] = [\ ] ++ X = X$$

для произвольного списка X.

Справедлива следующая *теорема о гомоморфизме списков* [60].

Теорема 1. Пусть заданы произвольная функция $\mathbf{f} : \mathcal{A} \to \mathcal{B}$ и произвольный ассоциативный оператор $\oplus : \mathcal{B} \times \mathcal{B} \to \mathcal{B}$. Тогда существует единственная функция $\mathbf{h} : [\mathcal{A}] \to \mathcal{B}$, удовлетворяющая следующим двум условиям:

$$\mathbf{h}.[a] = \mathbf{f}.a, \tag{12}$$

$$\mathbf{h}.(X ++ Y) = \mathbf{h}.X \oplus \mathbf{h}.Y \tag{13}$$

для всех $a : \mathcal{A}$, $X, Y : [\mathcal{A}]$.

Будем называть эту функцию *гомоморфизмом списков* и обозначать следующим образом: $[\![\mathbf{f},\oplus]\!]$. Таким образом, гомоморфизм списков $[\![\mathbf{f},\oplus]\!]$ всегда удовлетворяет условиям:

$$[\![\mathbf{f},\oplus]\!].[\cdot].a = \mathbf{f}.a, \tag{14}$$

$$[\![\mathbf{f},\oplus]\!].(X +\!\!+ Y) = [\![\mathbf{f},\oplus]\!].X \oplus [\![\mathbf{f},\oplus]\!].Y \tag{15}$$

для любых $a:\mathcal{A}$, $X,Y:[\mathcal{A}]$, и фактически является единственным решением этих уравнений. По сути, гомоморфизм – это замена $[\cdot]$ на $\mathbf{f}$ и $+\!\!+$ на $\oplus$, например

$$[\![\mathbf{f},\oplus]\!].\big([\cdot].a +\!\!+ [\cdot].b +\!\!+ [\cdot].c\big) = \mathbf{f}.a \oplus \mathbf{f}.b \oplus \mathbf{f}.c \tag{16}$$

для любых $a,b,c:\mathcal{A}$.

Отметим, что любая функция $\mathbf{h}:[\mathcal{A}]\to\mathcal{B}$, удовлетворяющая условию (13), по определению, является *гомоморфизмом* полугруппы $\big([\mathcal{A}],+\!\!+\big)$ в полугруппу $\big(\mathcal{B},\oplus\big)$. Утверждение, что полугруппа $\big([\mathcal{A}],+\!\!+\big)$ – свободная, эквивалентно утверждению, что функция $\mathbf{h}$ однозначно определяется ее значениями для одноэлементных списков.

Одним из простых примеров гомоморфизма является функция, возвращающая *длину* списка:

$$\#:[\mathcal{A}]\to\mathbb{Z}_{\geq 0}, \tag{17}$$

где $\mathbb{Z}_{\geq 0}$ обозначает множество целых неотрицательных чисел. Имеем:

$$\#.[a] = 1,$$

$$\#.\big(X +\!\!+ Y\big) = \#.X + \#.Y$$

для всех $a:\mathcal{A}$ и $X,Y:[\mathcal{A}]$. Отметим, что $\# = [\![\mathbf{k}_1,+]\!]$, где $\mathbf{k}_1:\mathcal{A}\to\mathbb{Z}_{\geq 0}$ и $\mathbf{k}_1.a = 1$ для всех $a:\mathcal{A}$.

Мультимножеством будем называть список, в котором порядок элементов не существенен. Для мультимножества операция конкатенации явля-

ется не только ассоциативной, но и коммутативной. Мультимножества будем обозначать с помощью пары символов $\lfloor$ и $\rfloor$. Операцию конкатенации мультимножеств будем обозначать с помощью символа $\uplus$ (объединение мультимножеств). Примерами мультимножеств целых неотрицательных чисел являются $\lfloor 1,2,2,5 \rfloor$ и $\lfloor 1,2,3,5 \rfloor$. Имеем

$$\lfloor 1,2,2,5 \rfloor \uplus \lfloor 1,2,3,4 \rfloor = \lfloor 1,2,3,4 \rfloor \uplus \lfloor 1,2,2,5 \rfloor = \lfloor 1,1,2,2,2,3,4,5 \rfloor.$$

Элементы мультимножеств в этом примере упорядочены по возрастанию для наглядности.

Конечным множеством (далее – *множеством*) называется такое мультимножество, в котором не допускаются повторения элементов. Операцию конкатенации мультимножеств будем обозначать с помощью символа $\cup$ (объединение мультимножеств). Для множества операция конкатенации не только коммутативна и ассоциативна, но и идемпотентна, то есть для любого множества A справедливо $A \cup A = A$. Множества обозначаются фигурными скобками. Примерами множеств целых неотрицательных чисел являются $\{1,2,5\}$ и $\{2,3,5\}$. Имеем $\{1,2,5\} \cup \{2,3,5\} = \{2,3,5\} \cup \{1,2,5\} = \{1,2,3,5\}$. Элементы множеств в этом примере упорядочены по возрастанию для наглядности. Теоретические построения, приводимые ниже, как правило, будут применимы не только к спискам, но и к множествам, и к мультимножествам.

В случае мультимножеств теорема о гомоморфизме означает, что для любой функции $\mathbf{f} : \mathcal{A} \to \mathcal{B}$ и произвольного ассоциативного и коммутативного оператора $\oplus : \mathcal{B} \to \mathcal{B}$, следующие два условия

$$\mathbf{h}.\lfloor a \rfloor = \mathbf{f}.a,$$

$$\mathbf{h}.(X \uplus Y) = \mathbf{h}.X \oplus \mathbf{h}.Y$$

однозначно определяют функцию $\mathbf{h}:\lfloor \mathcal{A} \rfloor \to \mathcal{B}$. Аналогичное утверждение справедливо и для множеств, за исключением того, что оператор $\oplus$ должен обладать свойством идемпотентности.

Простым примером гомоморфизма для мультимножеств является функция $\#$, возвращающая *размер* мультимножества (количество элементов, которое оно содержит, включая повторения). Имеем:

$$\#.\lfloor a \rfloor = 1,$$

$$\#.(\mathrm{X} \uplus Y) = \#.X + \#.\mathrm{Y}$$

для всех $a:\mathcal{A}$ и любых $X,Y:\lfloor \mathcal{A} \rfloor$. Таким образом функция $\#$ является гомоморфизмом коммутативной полугруппы $(\lfloor \mathcal{A} \rfloor, \uplus)$ в коммутативную полугруппу $(\mathbb{Z}_{\geq 0}, +)$. Отметим, что функция $\#$, вычисляющая *размер множества*, не является гомоморфизмом коммутативной полугруппы $(\{\mathcal{A}\}, \cup)$ в коммутативную полугруппу $(\mathbb{Z}_{\geq 0}, +)$, так как оператор $+$ не является идемпотентным в отличие от оператора $\cup$:

$$\#.(X) = \#.(X \cup X) \neq \#.X + \#.X = 2\#.X$$

для любого $X:\{\mathcal{A}\}$ и $X \neq \{\ \}$.

1.5.2. Оператор map

Основным оператором для работы со списками является бинарный *оператор* ***map*** обозначаемый с помощью символа $*$. Оператор **map** является функцией высшего порядка, так как его первым операндом является функция. В качестве второго операнда оператора **map** выступает список. Для функции $\mathbf{f}:\mathcal{A} \to \mathcal{B}$ функция $(\mathbf{f}*)$ имеет тип $[\mathcal{A}] \to [\mathcal{B}]$. С неформальной точки зрения функция $(\mathbf{f}*)$ применяет функцию $\mathbf{f}$ ко всем элементам списка, указанного в качестве второго аргумента оператора $*$:

$$\mathbf{f}*[a_1,...,a_n] = (\mathbf{f}*).[a_1,...,a_n] = [\mathbf{f}.a_1,...,\mathbf{f}.a_n].$$

Формально оператор **map** определяется следующими условиями:

$$(\mathbf{f}*).[\,] = [\,],$$

$$(\mathbf{f}*).[a] = [\mathbf{f}.a],$$

$$(\mathbf{f}*).[\cdot].a = [\cdot].\mathbf{f}.a,$$

$$(\mathbf{f}*).(X{+}{+}Y) = ((\mathbf{f}*).X){+}{+}((\mathbf{f}*).Y)$$

для всех $a:\mathcal{A}$ и $X,Y:[\mathcal{A}]$. Таким образом, функция $(\mathbf{f}*)$ является гомоморфизмом списков. Отметим, что $(\mathbf{f}*) = [\![[\cdot]\circ\mathbf{f},{+}{+}]\!]$. Также следует отметить, что оператор **map** обладает свойством дистрибутивности справа для операции композиции функций:

$$((\mathbf{f}\circ\mathbf{g})*) = (\mathbf{f}*)\circ(\mathbf{g}*).$$

1.5.3. Оператор reduce

Оператор **reduce** (reduction) представляет собой функцию высшего порядка, которая сворачивает список в одно значение путем многократного применения бинарного оператора, являющегося параметром оператора **reduce**. **Reduce** обозначается знаком / и имеет два аргумента: ассоциативный бинарный оператор и список. Пусть $\oplus:\mathcal{A}\times\mathcal{A}\to\mathcal{B}$ – произвольный ассоциативный оператор. Неформально оператор **reduce** можно определить следующим образом:

$$\oplus/[a_1,...,a_n] = a_1\oplus a_2\oplus...\oplus a_n.$$

Формально определим функцию $(\oplus/)$ с помощью следующих трех условий:

$$(\oplus/).[\,] = \mathbf{id}_{\oplus}, \tag{18}$$

$$(\oplus/).[a] = a, \tag{19}$$

$$(\oplus/).(X{+}{+}Y) = ((\oplus/).X)\oplus((\oplus/).Y) \tag{20}$$

для всех $a:\mathcal{A}$ и $X,Y:[\mathcal{A}]$. Таким образом, функция $(\oplus/)$ является гомоморфизмом списков. Отметим, что $(\oplus/)=[\![\mathbf{id},\oplus]\!]$. Если оператор $\oplus$ является не только ассоциативным, но и коммутативным, то функция $(\oplus/)$ будет гомоморфизмом мультимножеств; а если в добавок к этому $\oplus$ обладает свойством идемпотентности, то $(\oplus/)$ будет гомоморфизмом множеств.

Рассмотрим пример гомоморфизма списков с оператором **reduce**, действующим из полугруппы $([\mathbb{Z}_{\geq 0}],{+}{+})$ в полугруппу $(\mathbb{Z}_{\geq 0},+)$, определяемого условиями:

$$(+/).[a]=a, \tag{21}$$

$$(+/).(X{+}{+}Y)=((+/).X)+((+/).Y) \tag{22}$$

для всех $a:\mathbb{Z}_{\geq 0}$ и $X,Y:[\mathbb{Z}_{\geq 0}]$. С неформальной точки зрения функция $(+/)$ вычисляет сумму элементов списка целых неотрицательных чисел:

$$(+/).[a_1,...,a_n]=a_1+a_2+...+a_n.$$

Отметим, что оператор $+$ является коммутативным. Следовательно, функция $(+/)$ будет также гомоморфизмом мультимножеств. Однако, оператор $+$ не является идемпотентным. Поэтому функция $(+/)$ не будет гомоморфизмом множеств. Действительно, для любого непустого множества $X:\{\mathbb{Z}_{\geq 0}\}$ имеем $(+/).(X\cup X)=(+/).X=|X|$. С другой стороны, $(+/).X+(+/).X=|X|+|X|=2\cdot|X|\neq|X|$ (так как $X\neq\{\ \}$), что противоречит условию (6).

В качестве примера бинарного оператора, обладающего свойствами ассоциативности, коммутативности и идемпотентности, можно привести оператор $\uparrow$, вычисляющий максимум двух целых неотрицательных чисел. Имеем $(a\uparrow b)\uparrow c=a\uparrow(b\uparrow c)$, $a\uparrow b=b\uparrow a$ и $a\uparrow a=a$. Следовательно,

функция $(\uparrow /)$ является гомоморфизмом списков, мультимножеств и множеств.

1.6. Выводы по главе 1

Быстрый рост производительности вычислительных систем привел к фундаментальной смене парадигмы разработки численных алгоритмов. Если раньше необходимо было разрабатывать эффективные алгоритмы в условиях ограниченных вычислительных ресурсов, то сегодня необходимо разрабатывать алгоритмы, способные эффективно использовать большие вычислительные ресурсы. В соответствии с этим наиболее важной характеристикой численного алгоритма становится его граница масштабируемости, определяемая как максимум кривой, отражающей зависимость ускорения от количества процессорных узлов. Для оценки ускорения используются модели параллельных вычислений. Основными качественными характеристиками параллельной модели вычислений являются простота использования, универсальность и адекватность. Эти характеристики являются взаимно противоречивыми. Обзор известных моделей параллельных вычислений позволяет сделать вывод, что невозможно создать единую универсальную модель параллельных вычислений, хорошую во всех отношениях. Это связано с большим разнообразием и архитектурной сложностью современных многопроцессорных систем. В соответствии с этим для достижения приемлемого компромисса между простотой, универсальностью и адекватностью вычислительной модели необходимо сужать класс рассматриваемых аппаратных архитектур, ограничиваться алгоритмами определенного типа и накладывать более жесткие ограничения на структуру параллельных программ. Обзор моделей параллельных вычислений, приведенный в данной главе, опубликован в статье [176].

ГЛАВА 2. МОДЕЛЬ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ BSF

В данной главе описывается новая модель параллельных вычислений *BSF* (*Bulk Synchronous Farm*) – *блочно-синхронная ферма*, ориентированная на многопроцессорные системы с кластерной архитектурой и предназначенная для разработки итерационных численных алгоритмов с высокой вычислительной сложностью. Модель BSF предполагает представление алгоритма в виде операций над списками с использованием функций высшего порядка *Map* и *Reduce*. Отличительной особенностью модели BSF является то, что она позволяет получить аналитическую оценку границы масштабируемости алгоритма.

2.1. Базисные концепции

В основе модели BSF лежат следующие базисные концепции: схема программирования SPMD (Single Program – Multiple Data), парадигма «мастер-рабочие», модель параллельных вычислений BSP (Bulk-Synchronous Parallelism) и формализм Бёрда-Миртенса (Bird-Meertens formalism). Модель параллельных вычислений BSP была рассмотрена в разделе 1.3.3. Формализм Бёрда-Миртенса был детально описан в разделе 1.5.

Парадигма «*мастер-рабочие*» (*master-slave*) [100,139,142] является одним из популярных фреймворков, используемых в параллельном и распределенном программировании. В соответствии с этой парадигмой процессорные узлы вычислительной системы делятся на два множества: узлы-мастера и узлы-рабочие. Задача, решаемая на многопроцессорной системе, разбивается на независимые подзадачи. Каждая подзадача, в свою очередь, разбивается на три последовательные стадии: стадия предобработки, стадия вычислений и стадия постобработки. Стадии предобработки и постобработки выполняются узлами-мастерами, стадии вычислений выполняются

узлами-рабочими. В каждый момент времени любой процессорный узел может выполнять только одну стадию. Предполагается, что количество подзадач совпадает с количеством узлов-рабочих и превышает количество узлов-мастеров. Решение задачи происходит следующим образом. Очередная подзадача назначается некоторому узлу-мастеру, который выполняет стадию предобработки. После этого он посылает задание (передает данные) узлу-рабочему, который должен выполнить стадию вычислений указанной подзадачи. После того, как узел-рабочий завершил требуемые вычисления, он посылает отчет (передает данные) тому узлу-мастеру, от которого получил задание. На этом выполнение подзадачи заканчивается. Задача считается выполненной целиком, когда выполнены все ее подзадачи. Фреймворк «мастер-рабочие» очень часто используется в параллельном программировании при реализации различных приложений, ориентированных на многопроцессорные системы с распределенной памятью (см., например, [23,29,46,107,171]). При этом наиболее популярна конфигурация, включающая одного мастера и множество рабочих (см. рис. 16). Основной проблемой модели «мастер-рабочие» является нахождение такого расписания вычислений, при котором время выполнения задачи будет минимальным. Известно, что указанная проблема является в общем случае NP-сложной [138]. В определенной мере эту проблему можно решить, используя схему программирования *SPMD*.

SPMD (*Single Program Multiple Data*) [44,142] – популярная схема параллельного программирования, в соответствии с которой все процессорные узлы выполняют одну и ту же программу, но обрабатывают различные данные. Какие именно данные необходимо обрабатывать тому или иному процессорному узлу, определяется его уникальным номером, который является параметром программы. Данный подход наиболее часто используется в сочетании с технологией MPI (Message Passing Interface) [72], которая де-факто является стандартом для параллельного программирования на распределенной памяти.

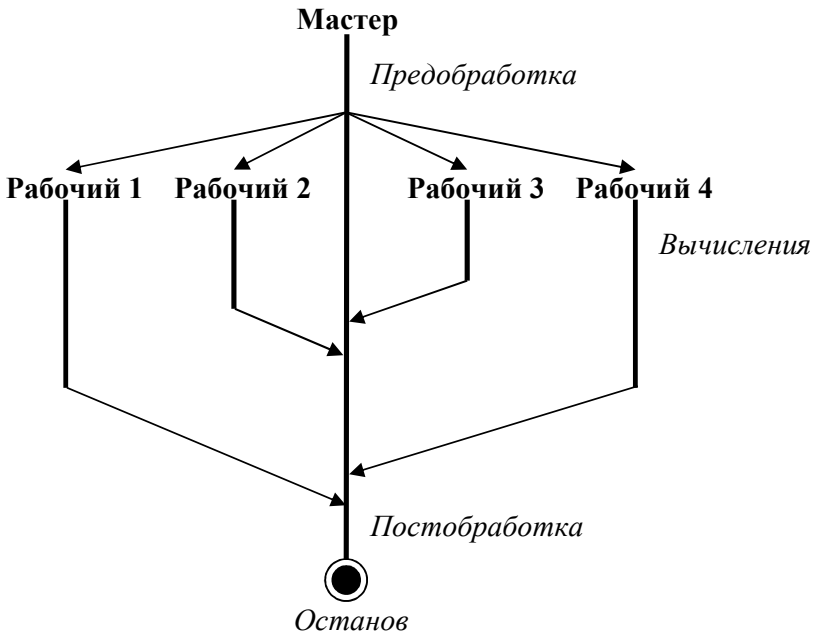

Рис. 16. Фреймворк «мастер-рабочие». Полужирными линиями обозначены потоки вычислений, тонкими линями со стрелками – потоки данных.

При разработке параллельного алгоритма для многопроцессорных систем с распределенной памятью одной из самых важных характеристик является масштабируемость. *Масштабируемость* можно определить как способность алгоритма эффективно использовать увеличивающиеся вычислительные ресурсы, главным из которых является количество доступных процессорных узлов. Основной величиной для определения уровня масштабируемости алгоритма является *ускорение*. Ускорение a как функция, зависящая от количества K процессорных узлов, определяется следующей формулой:

$$a(K) = \frac{T_1}{T_K}, \tag{23}$$

где T_1 – время решения задачи на одном процессорном узле, T_K – время решения задачи на K процессорных узлах. Для определения границы масштабируемости необходимо исследовать кривую, отражающую зависимость ускорения от количества процессорных узлов. На рис. 17 изображены

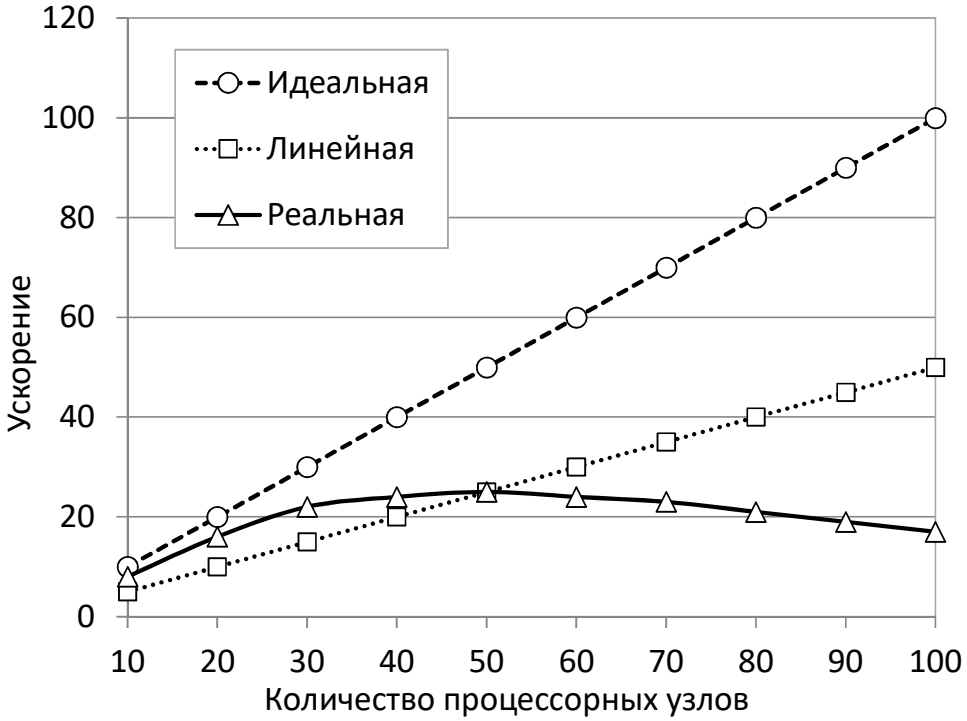

Рис. 17. Различные формы кривой ускорения.

различные формы кривой ускорения. Ускорение называется *идеальным*, если оно прямо пропорционально количеству процессорных узлов: $a(K) = K$. Ускорение называется *линейным*, если наблюдается линейная зависимость величины ускорения: $a(K) = \lambda K$, где $0 < \lambda < 1$. *Реальному* ускорению на практике наиболее часто соответствует кривая, которая монотонно возрастает до некоторого количества процессорных узлов (50 на рис. 17), после чего начинается деградация. Количество процессорных узлов, на котором достигается максимальное ускорение в этом случае называется *границей масштабируемости алгоритма* (см. рис. 18). При разработке параллельных алгоритмов для больших многопроцессорных систем с распределенной памятью очень важно иметь возможность оценить границу масштабируемости алгоритма на ранней стадии его разработки.

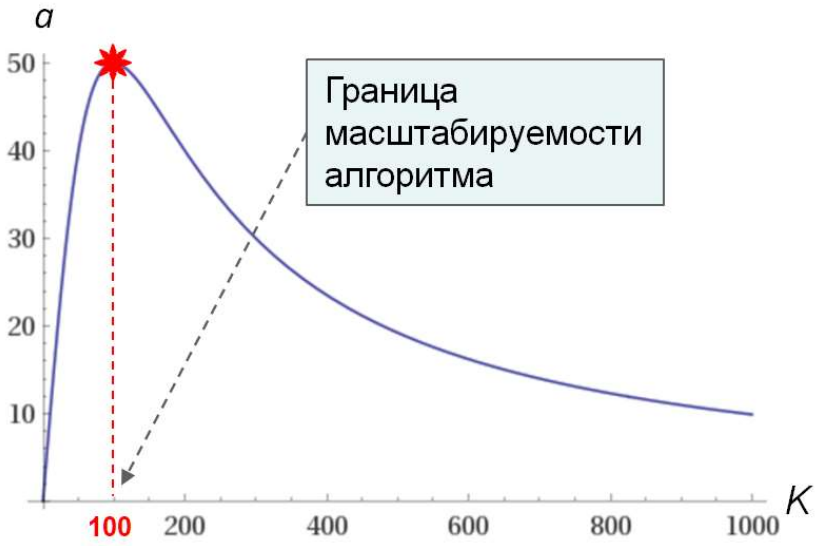

Рис. 18. Кривая ускорения и граница масштабируемости алгоритма.

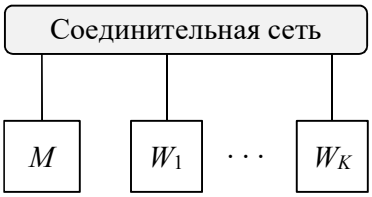

Рис. 19. *BSF*-компьютер. М –мастер;
W_1, …, W_K – рабочие.

2.2. BSF-компьютер и BSF-алгоритм

BSF-компьютер представляет собой множество однородных процессорных узлов с приватной памятью, соединенных сетью, позволяющей передавать данные от одного процессорного узла другому. Среди процессорных узлов выделяется один, называемый *узлом-мастером* (или кратко *мастером*). Остальные K узлов называются *узлами-рабочими* (или просто *рабочими*). В *BSF*-компьютере должен быть по крайней мере один узел мастер и один рабочий ($K \geq 1$). Схематично архитектура *BSF*-компьютера изображена на рис. 19.

Алгоритм 2. Шаблон итерационного алгоритма в модели BSF

1: **input** $A, x^{(0)}$
2: $i := 0$
3: $B := Map\left(F_{x^{(i)}}, A\right)$
4: $s := Reduce(\oplus, B)$
5: $x^{(i+1)} := Compute\left(x^{(i)}, s\right)$
6: $i := i + 1$
7: **if** $StopCond(x^{(i)}, x^{(i-1)})$ **goto** 9
8: **goto** 2
9: **output** $x^{(i)}$
10: **stop**

Рис. 20. Шаблон итерационного алгоритма в модели BSF.

BSF-алгоритм должен быть представлен в виде операций над списками с использованием функций высшего порядка *Map* и *Reduce*, определяемым в формализме Бёрда-Миртенса (см. раздел 1.5). *Общий шаблон итерационного алгоритма* в модели BSF представлен на рис. 20. Переменная i обозначает номер итерации; $x^{(0)}$ – начальное приближение, $x^{(i)}$ – i-тое приближение (в качестве приближения может фигурировать число, вектор или другая произвольная структура данных); A – список элементов некоторого множества $\mathbb{A}$, представляющий собой исходные данные задачи; $F_x : \mathbb{A} \to \mathbb{B}$ – параметризованная функция (в качестве параметра фигурирует приближение x), отображающая множество $\mathbb{A}$ в некоторое множество $\mathbb{B}$; B – список элементов множества $\mathbb{B}$, получающийся путем применения функции F_x к каждому элементу списка A; $\oplus$ – некоторая бинарная ассоциативная операция над множеством $\mathbb{B}$. На шаге 1 вводятся исходные данные задачи (список A) и начальное приближение $x^{(0)}$. На шаге 2 номер итерации устанавливается в значение 0. На шаге 3 вычисляется список B как результат выполнения функции высшего порядка $Map(F_{x^{(i)}}, A)$. На шаге 4 вычисляется промежуточная переменная $s \in \mathbb{B}$, которая получается в результате выполнения

Алгоритм 3. Шаблон параллельной реализации итерационного алгоритма в модели BSF

Мастер	j-ый рабочий (j=1,…,K)
1: **input** $x^{(0)}$	1: **input** $A^{[j]}$
2: $i := 0$	2:
3: $SendToAllWorkers(x^{(i)})$	3: $RecvFromMaster(x^{(i)})$
4:	4: $B^{[j]} := Map(F_{x^{(i)}}, A^{[j]})$
5:	5: $s^{(j)} := Reduce(\oplus, B^{[j]})$
6: $RecvFromWorkers([s^{(1)}, \dots, s^{(K)}])$	6: $SendToMaster(s^{(j)})$
7: $s := Reduce(\oplus, [s^{(1)}, \dots, s^{(K)}])$	7:
8: $x^{(i+1)} := Compute(x^{(i)}, s)$	8:
9: $i := i + 1$	9:
10: **if** $StopCond(x^{(i)}, x^{(i-1)})$ **goto** 12	10:
11: **goto** 3	11: **goto** 3
12: **output** x_i	12:
13: **stop**	13:

Рис. 21. Шаблон параллельной реализации итерационного алгоритма в модели BSF.

функции высшего порядка $Reduce(\oplus, B)$. На шаге 5 выполняется пользовательская функция *Compute*, которая на основе $x^{(i)}$ и s вычисляет следующее приближение $x^{(i+1)}$. На шаге 6 номер итерации i увеличивается на единицу. На шаге 7 выполняется пользовательская булева функция *StopCond* с аргументами $x^{(i)}$ и $x^{(i-1)}$, проверяющая условие завершения работы итерационного алгоритма. Если эта функция возвращает значение «истина», то происходит переход на шаг 9. На шаге 8 осуществляется безусловный переход на шаг 3 для выполнения очередной итерации. На шаге 9 в качестве результата выводится последнее полученное приближение. Шаг 10 останавливает работу алгоритма.

Общий шаблон параллельной реализации итерационного алгоритма модели BSF представлен на рис. 21. Параллельная реализация включает в себя $K+1$ потоков управления. Поток управления с номером 0 выполняется на

узле-мастере. Потоки $1,\ldots,K$ выполняются на узлах-рабочих, имеют идентичный код, однако обрабатывают различные части $A^{[1]},\ldots,A^{[K]}$ списка A, формируя различные части $B^{[1]},\ldots,B^{[K]}$ списка B. Деление списка A на подсписки $A^{[1]},\ldots,A^{[K]}$ осуществляется таким образом, чтобы все они, возможно кроме последнего, имели равную длину. На шаге 3 мастер посылает, а рабочие получают текущее приближение $x^{(i)}$. После этого *j*-тый рабочий выполняет следующие действия: на шаге 4 вычисляет подсписок $B^{[j]}$, применяя функцию $F_{x^{(i)}}$ к каждому элементу подсписка $A^{[j]}$; на шаге 5 считает частичную «сумму» $s^{(j)}$, «складывая» элементы подсписка $B^{[j]}$ с помощью операции $\oplus$; на шаге 6 посылает вычисленное значение $s^{(j)}$ мастеру. Отметим, что указанные шаги зависят только от локальных данных и выполняются всеми рабочими параллельно. При этом время, затрачиваемое рабочими с номерами $1,\ldots,K-1$ на выполнение шагов 4 и 5, будет одинаково, так как они обрабатывают списки равной длины. K-тый рабочий будет работать меньшее время, если длина исходного списка не кратна K. Мастер на шаге 6 получает значения $s^{(j)}$ от всех рабочих и выполняет оставшиеся шаги итерационного алгоритма. Отметим, что для корректной работы параллельного алгоритма 3 необходимо и достаточно, чтобы операция $\oplus$ была ассоциативной. Это следует из теоремы о гомоморфизме списков (см. раздел 1.5). В дальнейшем список A, являющийся аргументом функции Map, будем называть *списком* «*Map*», а список B, являющийся аргументом функции *Reduce*, будем называть *списком* «*Reduce*».

Графическая иллюстрация работы *BSF*-программы приведена на рис. 22. В начале текущей итерации мастер пересылает всем рабочим данные текущей итерации. Рабочие выполняют шаги «*Map*» и «*Reduce*» для своей части исходного списка. Поскольку список делится на равные части по числу

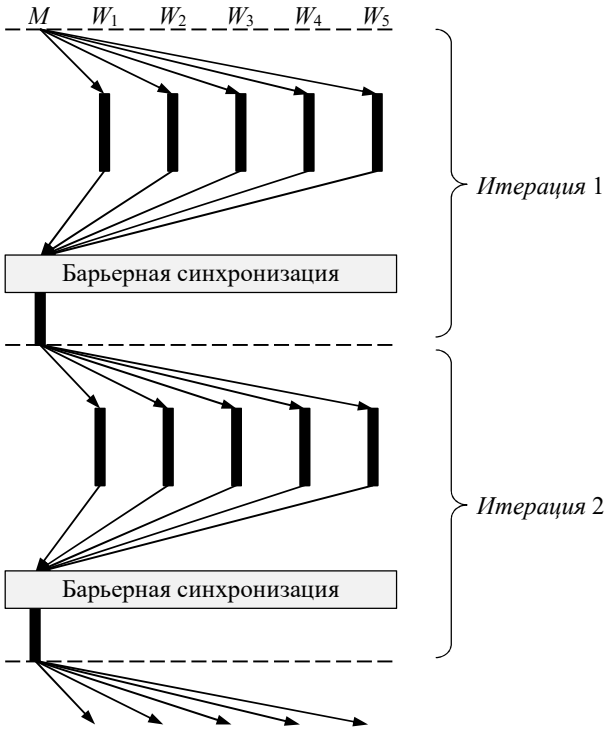

Рис. 22. Иллюстрация работы *BSF*-программы
с одним мастером M и пятью рабочими $W_1,\ldots,W_5$
(жирными линиями обозначены локальные вычисления,
тонкими линями со стрелками – пересылка данных,
пунктирными линиями – границы одной итерации).

рабочих, и рабочие выполняют один и тот же код, время их работы на однородных процессорных узлах будет одинаковым. Это снимает проблему балансировки загрузки процессорных узлов. Во время вычислений, производимых рабочими, мастер простаивает. По завершению вычислений рабочие направляют свои частичные результаты мастеру. Мастер не начинает их обработку, пока результаты не будут получены от всех рабочих. Тем самым выполняется барьерная синхронизация. После этого мастер выполняет *Reduce* над частичными результатами, полученными от рабочих, вычисляет очередное приближение и, если оно не удовлетворяет условию завершения, переходит к следующей итерации. Во время вычислений, производимых мастером, рабочие простаивают.

2.3. Стоимостная метрика модели BSF

Мы предполагаем, что временные затраты на инициализацию и завершение *BSF*-программы пренебрежимо малы по сравнению с затратами на выполнение итерационного процесса. Стоимость итерационного процесса получается как сумма стоимостей отдельных итераций. Поэтому для оценки времени выполнения *BSF*-программы нам достаточно получить оценку временной стоимости одной итерации. Модель BSF включает в себя следующие основные стоимостные параметры в рамках одной итерации:

K – количество узлов-рабочих;

t_s – время, затрачиваемое мастером на передачу задания одному рабочему (без учета латентности);

t_{Map} – время выполнения одним рабочим функции *Map* для *всего* списка исходных данных A;

t_p – время, затрачиваемое мастером на обработку полученных результатов и проверку условия завершения;

t_r – время, необходимое для передачи мастеру результата, полученного одним рабочим (без учета латентности);

t_a – время, необходимое для выполнения одной операции $\oplus$, являющейся параметром функции *Reduce*;

l – длина списка исходных данных A (совпадает с длиной списка B);

L – латентность (время пересылки сообщения длиной в 1 байт).

Обозначим через T_1 время выполнения одной итерации алгоритма 3 (рис. 21) системой из одного мастера и одного рабочего (см. рис. 23). Используя введенные стоимостные параметры, можно получить следующую оценку времени T_1:

$$T_1 = 2L + t_s + t_r + t_p + t_{Map} + (l-1)t_a. \qquad (24)$$

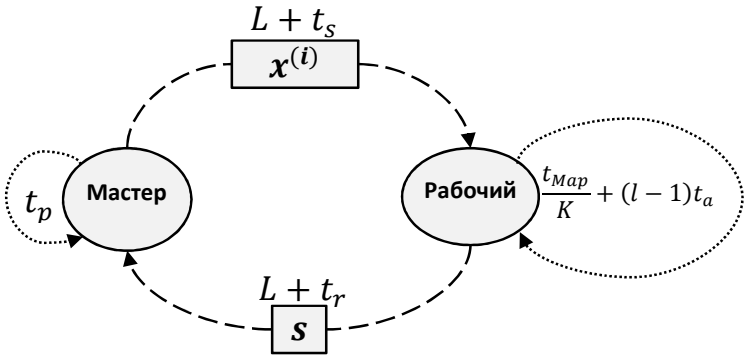

Рис. 23. Схема работы алгоритма 3 (рис. 21) для конфигурации из одного мастера и одного рабочего.

При $l \to \infty$ формула (24) асимптотически приближается к следующей оценке:

$$T_1 = 2L + t_s + t_r + t_p + t_{Map} + lt_a. \tag{25}$$

Обозначим через T_K время выполнения одной итерации алгоритма 3 системой из одного мастера и K рабочих (см. рис. 24). Для времени T_K можно получить следующую оценку:

$$T_K = K\left(2L + t_s + t_r + t_a\right) + \frac{\left(t_{Map} + \left(l-1\right)t_a\right)}{K} - t_a + t_p. \tag{26}$$

При длине списка $l \to \infty$ формула (26) асимптотически приближается к следующей оценке:

$$T_K = K\left(2L + t_s + t_r + t_a\right) + \frac{t_{Map} + lt_a}{K} - t_a + t_p. \tag{27}$$

Ускорение a как функция от K в модели BSF вычисляется по формуле:

$$a(K) = \frac{T_1}{T_K} = \frac{2L + t_s + t_r + t_p + t_{Map} + lt_a}{K\left(2L + t_s + t_r + t_a\right) + \frac{t_{Map} + lt_a}{K} - t_a + t_p}. \tag{28}$$

При положительных значениях всех параметров и при $K \geq 1$ функция, определяемая формулой (28), обладает следующими свойствами:

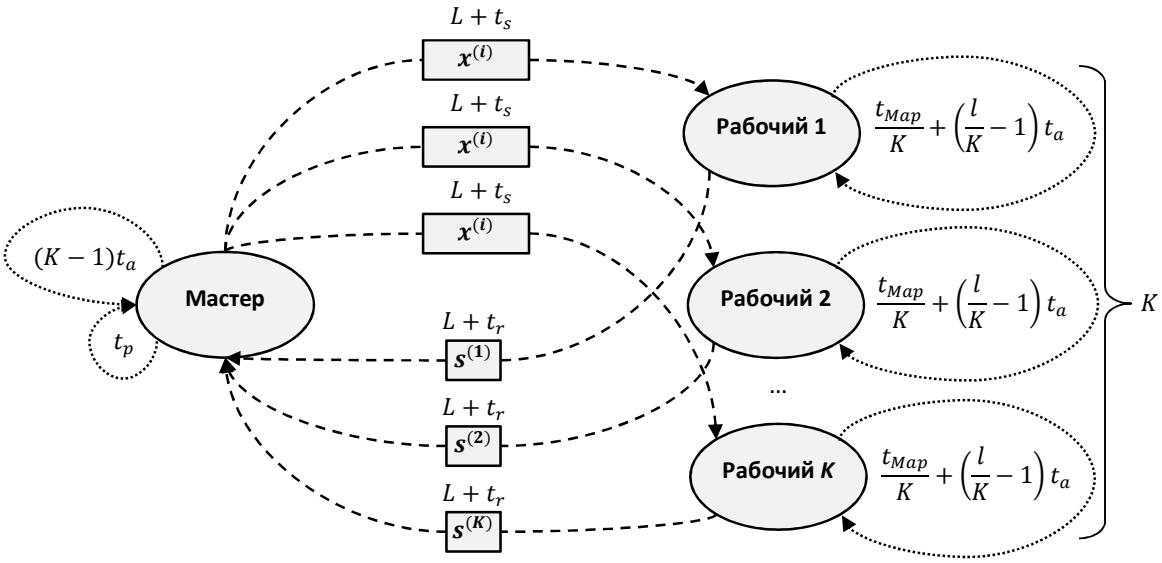

Рис. 24. Схема работы 3 для конфигурации из одного мастера и K рабочих.

1) $a(1) = 1$;

2) $a(K) > 0$;

3) $\lim a(K) = \frac{1}{K}$.

Все указанные свойства непосредственно следуют из формулы (28) и не требуют доказательства. С содержательной точки зрения свойство 1) соответствует реальности: ускорение на одном процессорном узле должно быть равно 1. Свойство 2) также подтверждает адекватность формулы (28), так как ускорение – всегда величина положительная. Свойство 3) говорит, что при очень малых значениях параметров t_{Map}, t_a и t_p, определяющих суммарный объем вычислений, формула (28) на интервале $[1;+\infty)$ вырождается в монотонно убывающую функцию $a(K) = 1/K$, не имеющую максимума внутри указанного интервала. Это означает, что модель BSF не применима для алгоритмов, в которых время, затрачиваемое на пересылку данных, несоизмеримо больше времени, затрачиваемого на вычисления. Главное свойство формулы (28) сформулируем в виде следующей *теоремы о границе масштабируемости BSF-алгоритма*.

Теорема 2. Пусть все параметры в формуле (28) положительны. Тогда функция $a(K)$, определяемая формулой (28), имеет единственный экстремум на интервале $(1;+\infty)$, являющийся максимумом.

Доказательство. Для нахождения точек экстремума функции (28) найдем производную ускорения по K:

$$a'(K)=\frac{\left(2L+t_s+t_r+t_p+t_{Map}+lt_a\right)\cdot\left(-2L-t_s-t_r-t_a+\frac{t_{Map}+lt_a}{K^2}\right)}{\left(K\left(2L+t_s+t_r+t_a\right)+\frac{t_{Map}+lt_a}{K}-t_a+t_p\right)^2}. \quad (29)$$

Уравнение

$$\frac{\left(2L+t_s+t_r+t_p+t_{Map}+lt_a\right)\cdot\left(-2L-t_s-t_r-t_a+\frac{t_{Map}+lt_a}{K^2}\right)}{\left(K\left(2L+t_s+t_r+t_a\right)+\frac{t_{Map}+lt_a}{K}-t_a+t_p\right)^2}=0. \quad (30)$$

имеет только один корень на интервале $[K,+\infty)$:

$$K_0=\sqrt{\frac{t_{Map}+lt_a}{2L+t_s+t_r+t_a}}. \quad (31)$$

Несложно увидеть, что на интервале $[1,K_0)$ производная $a'(K)$, вычисляемая по формуле (29), принимает только положительные значения, а на интервале $(K_0,+\infty)$ – только отрицательные. Следовательно, значение $a(K)$ достигает максимума в точке K_0. Теорема доказана.

Таким образом мы получаем, что *граница масштабируемости BSF-алгоритма* может быть оценена по следующей формуле:

$$K_{MAX}=\sqrt{\frac{t_{Map}+lt_a}{2L+t_s+t_r+t_a}}. \quad (32)$$

2.4. Выводы по главе 2

Во второй главе дано описание новой модели параллельных вычислений BSF. Модель BSF ориентирована на итерационные алгоритмы с высокой вычислительной сложностью, выполняемые на больших вычислительных кластерах. Описаны все основные компоненты модели, перечисленные в разделе 1.2. Модель BSF требует представления алгоритма в виде операций над списками с использованием функций высшего порядка *Map* и *Reduce*. Построена стоимостная метрика, на основе которой получена аналитическая формула для оценки ускорения BSF-алгоритма. Доказана теорема о границе масштабируемости BSF-алгоритма. Указанная теорема дает формулу для аналитической оценки верхней границы масштабируемости. Ни одна из известных моделей параллельных вычислений такой оценки получить не позволяет. Результаты этой главы опубликованы в работах [174,177,178].

ГЛАВА 3. ПРОГРАММНАЯ ПОДДЕРЖКА МОДЕЛИ BSF

В данной главе, в разделе 3.1 описывается структура параллельного BSF-каркаса на языке C++, используемого для быстрого создания BSF-программ. В разделе 3.2 описывается визуальный конструктор BSF-программ.

3.1. BSF-каркас

Для быстрого создания BSF-программ на языке C++ в рамках диссертационного исследования был разработан параллельный BSF-каркас, ориентированный на кластерные вычислительные системы. Для организации обменов сообщениями между процессорными узлами используется программный интерфейс MPI [70–72]. BSF-каркас спроектирован таким образом, что он допускает поэтапное заполнение проблемно-зависимых частей, и при этом компилируется на всех этапах разработки. Исходные тексты BSF-каркаса свободно доступны в сети Интернет по адресу https://github.com/nadezhda-ezhova/BSF-MR. Заполненный BSF-каркас компилируется и компонуется вместе с библиотеками MPI и OpenMP в один исполняемый файл, который запускается как MPI-процесс на указанном количестве узлов в указанном количестве экземпляров.

3.1.1. Файловая структура каркаса

BSF-каркас состоит из двух групп файлов:

1) файлы с префиксом «BSF» содержат проблемно-независимый код и не подлежат изменениям со стороны пользователя;
2) файлы с префиксом «Problem» предназначены для заполнения пользователем проблемно-зависимых частей программы.

Табл. 4. Файлы исходного кода BSF-каркаса.

Файл	Описание
Проблемно-независимый код	
BSF-Code.cpp	Реализации головной функции main и всех проблемно-независимых функций
BSF-Data.h	Проблемно-независимые переменные и структуры данных
BSF-Forwards.h	Предописания проблемно-независимых функций
BSF-Include.h	Включение проблемно-независимых библиотек (iostream, mpi.h, omp.h)
BSF-ProblemFunctions.h	Предописания предопределенных функций с проблемно-зависимой реализацией
BSF-Types.h	Определения проблемно-независимых типов
Проблемно-зависимый код	
Problem-bsfCode.cpp	Реализация предопределенных проблемно-зависимых функций
Problem-bsfParameters.h	Предопределенные проблемно-зависимые параметры
Problem-bsfTypes.h	Предопределенные проблемно-зависимые типы
Problem-Data.h	Проблемно-зависимые переменные и структуры данных
Problem-Forwards.h	Предописания проблемно-зависимых функций
Problem-Include.h	Включение проблемно-зависимых библиотек
Problem-Parameters.h	Проблемно-зависимые параметры
Problem-Types.h	Проблемно-зависимые типы

Описания всех файлов исходного кода приведены в табл. 4. Граф зависимостей файлов BSF-каркаса по включению с помощью директивы #include приведен на рис. 25. Серым закрашены файлы, в которых пользовательские изменения не допускаются; узорной штриховкой обозначены файлы с предопределенными заголовками определений, тело которых должен заполнить

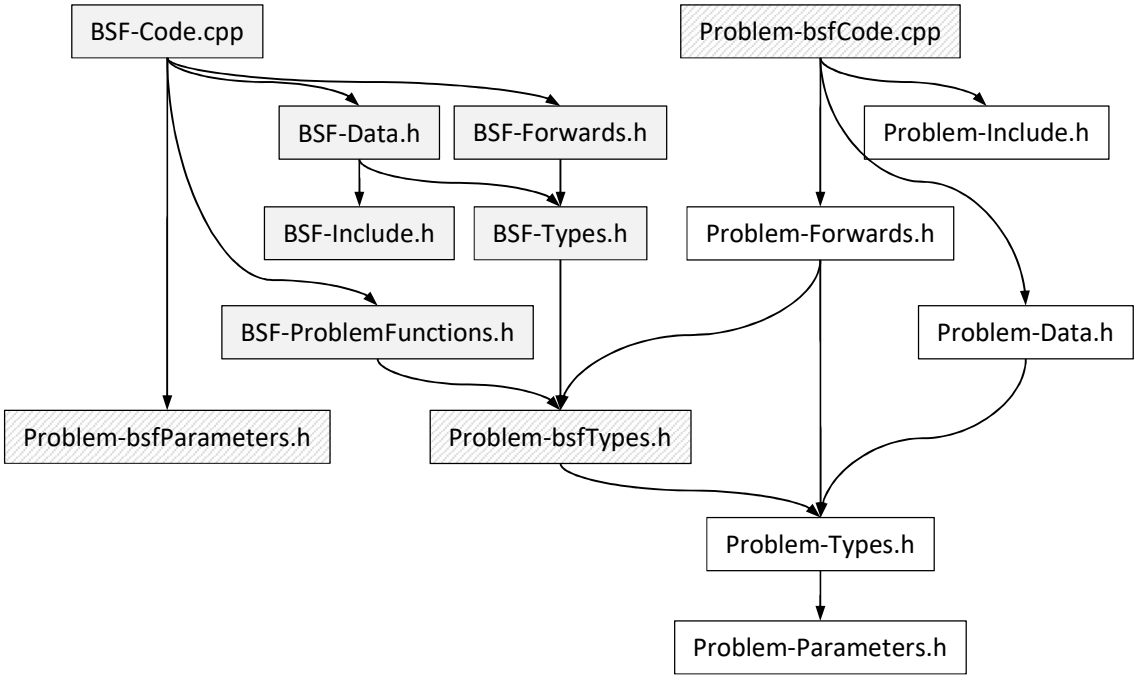

Рис. 25. Граф зависимостей файлов BSF-каркаса по включению #include.

пользователь; белый фон соответствует файлам, полностью заполняемым пользователем. Порядок заполнения BSF-каркаса приведен в файле ReadMe.pdf, доступном по адресу https://github.com/nadezhda-ezhova/BSF-MR.

Файл ***Problem-bsfParameters.h*** содержит следующие предопределенные константы (#define), используемые в BSF-Code.cpp, значения которых устанавливает пользователь:

– *PP_BSF_PRECISION*: задает количество значащих цифр (ширину поля вывода) для чисел с плавающей точкой;
– *PP_BSF_ITER_OUTPUT*: если указанный #define определен, то будет выполняться вывод результатов итераций с определенным шагом, задаваемым константой *PP_BSF_TRACE_COUNT*;
– *PP_BSF_TRACE_COUNT*: задает частоту вывода результатов итераций, например, если указано значение 10, то будет выводиться результат каждой десятой итерации (если определен #define *PP_BSF_ITER_OUTPUT*);

– *PP_BSF_OMP*: если указанный #define определен, то будет включена #pragma omp parallel for перед циклом, реализующим функцию Map;
– *PP_BSF_NUM_THREADS*: указывает количество потоков управления, в директиве #pragma omp parallel (если константа *PP_BSF_NUM_THREADS* не определена, то запускается количество потоков, определенное по умолчанию).

Файл ***Problem-bsfTypes.h*** включает в себя заголовки описаний (описания с пустым телом) трех предопределенных структурных типов:

– *PT_bsf_data_T*: описывает структуру данных, передаваемых каждому рабочему для выполнения очередной итерации (может содержать текущее приближение и другие параметры);
– *PT_bsf_mapElem_T*: описывает структуру элемента списка «Map»;
– *PT_bsf_reduceElem_T*: описывает структуру элемента списка «Reduce».

Файл ***BSF-ProblemFunctions.h*** включает в себя предописания (форварды) следующих предопределенных пользовательских функций, которые должны быть реализованы в файле *Problem-bsfCode.cpp*:

– *PC_bsf_AssignListSize(int* listSize)*: присваивает переменной **listSize* длину списка «Map» (совпадает с длиной списка «Reduce»);
– *PC_bsf_CopyData(PT_bsf_data_T* dataIn, PT_bsf_data_T* dataOut)*: копирует данные из структуры **dataOut* в структуру **dataIn*;
– *PC_bsf_IterOutput(PT_bsf_reduceElem_T* reduceResult, int count, PT_bsf_data_T data, int iterCount, double elapsedTime)*: выводит результаты итерации, используя в качестве параметров *reduceResult* – результат выполнения функции Reduce над всем списком; *count* – ко-

личество элементов, участвовавших в Reduce; *data* – задание, выполненное на данной итерации; *iterCount* – количество выполненных итераций; *elapsedTime* – общее время, затраченное на решение задачи.

- *PC_bsf_Init(bool* success)*: выделяет память для проблемно-зависимых структур данных и выполняет их инициализацию; если необходимую память выделить не удалось, переменной **success* должно быть присвоено значение *false*;
- *PC_bsf_MapF(PT_bsf_mapElem_T* mapElem, PT_bsf_reduceElem_T* reduceElem, PT_bsf_data_T* data, int* success)*: вычисляет функцию *F*, являющуюся параметром оператора *Map*, используя аргументы **mapElem* – элемент списка «*Ma*р», над которым выполняется функция *F*; **reduceElem* – элемент списка «*Reduce*», которому должно быть присвоено значение функции *F*; **data* – задание, содержащее текущее приближение; параметру **success* должно быть присвоено значение 0, если полученный элемент списка «*Reduce*» должен игнорироваться при выполнении операции *Reduce*.
- *PC_bsf_ParametersOutput(int numOfWorkers, PT_bsf_data_T data)*: выводит начальные параметры задачи, используя аргументы *numOfWorkers* – количество процессов-рабочих и *data* – начальное задание, содержащее начальное приближение;
- *PC_bsf_ProblemOutput(PT_bsf_reduceElem_T* reduceResult, int count, PT_bsf_data_T data, int iterCount, double t)*: выводит конечные результаты работы программы, используя аргументы **reduceResult* – результат выполнения оператора Reduce, *count* количество элементов «суммированных» при выполнении оператора Reduce, *data* – последнее задание (последнее приближение), *t* – общее время работы программы;
- *PC_bsf_ProcessResults(bool* exit, PT_bsf_reduceElem_T* reduceResult, int count, PT_bsf_data_T* data)*: обрабатывает результаты, полученные

в результате выполнения очередной итерации, используя аргументы **reduceResult* – результат выполнения оператора Reduce, *count* количество элементов «суммированных» при выполнении оператора Reduce, *data* – последнее задание (последнее приближение); если вычисления необходимо продолжить, переменной **exit* присваивается значение *true*, в противном случае – *false*;

- *PC_bsf_ReduceF(PT_bsf_reduceElem_T* x, PT_bsf_reduceElem_T* y, PT_bsf_reduceElem_T* z)*: реализует функцию, являющуюся аргументом оператора Reduce, по формуле $z := x \oplus y$.
- *PC_bsf_SetInitApproximation(PT_bsf_data_T* data)*: записывает в **data* начальное задание (начальное приближение);
- *PC_bsf_SetMapSubList(PT_bsf_mapElem_T* subList, int count, int offset, bool* success)*: заполняет подсписок списка «Map», используя аргументы **subList* – указатель на первый элемент подсписка, *count* – длина подсписка, *offset* – сдвиг, относительно начала списка; если у элемента списка «Map» имеются поля типа указатель, то необходимо выделить память под вектор, матрицу или другую структуру; если обнаружена нехватка памяти, переменной **success* необходимо присвоить значение *false*.

Файл ***BSF-Types.h*** содержит определения двух проблемно-независимых структурных типов: *BT_order_T* и *BT_extendedReduceElem_T*. Структурный тип *BT_order_T* определяет структуру задания, пересылаемого мастером рабочим, и включает в себя два поля: *data* типа *PT_bsf_data_T*, определяемого в файле *Problem-bsfTypes.h* (см. стр. 89), и *exit* типа *char*. Если поле *exit* содержит двоичное значение 1, то процессы-рабочие должны завершить свою работу. Структурный тип *BT_extendedReduceElem_T* определяет структуру элемента расширенного списка «*Reduce*» и включает в себя два поля: *elem* типа *PT_bsf_reduceElem_T*, определяемого в файле *Problem-bsfTypes.h*

(см. выше), и *counter* типа *int*. До выполнения оператора *Reduce* в поле *counter* может находиться значение 1, либо 0. Значение 0 означает, что данный элемент должен быть проигнорирован при выполнении оператора *Reduce*. Значение 1 означает, что данный элемент должен быть учтен при выполнении оператора *Reduce*. После выполнения оператора *Reduce* над списком (подсписком) в поле *counter* результирующего элемента заносится количество учтенных исходных элементов.

Файл ***BSF-Data.h*** содержит определения проблемно-независимых переменных и структур данных. В число переменных входят следующие:

- *BD_listSize*: длина списка «Map» (совпадает с длиной списка «Reduce»);
- *BD_size*: количество MPI-процессов;
- *BD_rank*: номер текущего MPI-процесса;
- *BD_masterRank*: номер процесса-мастера (равен $BD\_size - 1$);
- *BD_numOfWorkers*: количество процессов-рабочих (равно $BD\_size - 1$);
- *BD_elemsPerWorker*: количество элементов списка, приходящихся на одного рабочего (равно $\lfloor BD\_listSize \,/\, BD\_numOfWorkers \rfloor$;
- *BD_tailLength*: длина остатка списка после деления на число рабочих (равно $BD\_listSize - BD\_elemsPerWorker \cdot BD\_numOfWorkers$);
- *BD_exit*: индикатор завершения вычислений;
- *BD_success*: индикатор успешного выполнения инициализации;
- *BD_t*: время, затраченное на вычисления (не учитываются ввод/вывод данных и инициализация);
- *BD_iterCount*: количество выполненных итераций.

В число структур данных входят следующие:

- *BD_data*: структура, в которой формируется данные, необходимые рабочим для выполнения очередной итерации;

- *BD_mapSubList*: указатель на подсписок, обрабатываемый рабочим (используется только процессами-рабочими);
- *BD_extendedReduceList*: указатель на расширенный список «Reduce»;
- *BD_extendedReduceResult_P*: указатель на структуру, в которой вычисляется результирующий элемент при выполнении оператора Reduce над расширенным списком «Reduce»;
- *BD_order*: указатель на структуру, в которой формируется задание для рабочего;
- *BD_status*: указатель на массив системных MPI-структур «MPI_Status», содержащих параметры сообщений для каждого рабочего;
- *BD_request*: указатель на массив системных MPI-переменных, содержащий идентификаторы асинхронных обменов по числу процессов-рабочих (используется только процессом-мастером);
- *BD_subListSize*: указатель на массив целых чисел, содержащий размеры подсписков для всех рабочих;
- *BD_offset*: указатель на массив целых чисел, содержащий для каждого рабочего сдвиг его подсписка относительно начала общего списка.

Файл ***BSF-Code.cpp*** включает в себя реализацию головной функции main и реализации следующих проблемно-независимых функций:

- *BC_MpiRun* выполняет инициализацию MPI и соответствующих переменных;
- *BC_Init выделяет* необходимую память для структур данных, инициализирует переменные и заполняет список «Map» исходными данными;
- *BC_Master* реализует процесс, выполняемый на узле-мастере;
- *BC_Worker* реализует процесс, выполняемый на узлах-рабочих;
- *BC_MasterMap* реализует шаг «Map» на узле-мастере;

– *BC_MasterReduce* реализует шаг «Reduce» на узле-мастере;
– *BC_MpiRun* осуществляет инициализацию MPI;
– *BC_WorkerMap* реализует шаг «Map» на узлах-рабочих;
– *BC_WorkerReduce* реализует шаг «Reduce» на узлах-рабочих;
– *BC_ProcessExtendedReduceList* обрабатывает указанную часть списка «Reduce».

Файл ***BSF-Include.h*** подключает с помощью директивы #include необходимые системные библиотеки. Файл ***BSF-Forwards.h*** содержит предописания функций, реализуемых в файле *BSF-Code.cpp.* Файл ***Problem-bsfCode.cpp*** содержит реализации проблемно-зависимых функций, предописания которых находятся в файле *BSF-ProblemFunctions.h* (см. стр. 89). Файл ***Problem-Include.h*** подключает с помощью директивы #include системные библиотеки, необходимые для реализаций проблемно-зависимых функций. Файл ***Problem-Forwards.h*** содержит предописания пользовательских функций, используемых для реализации предопределенных проблемно-зависимых функций. Файл ***Problem-Types.h*** содержит определения пользовательских типов данных. Файл ***BSF-Types Problem-Data.h*** содержит определения пользовательских глобальных переменных и структур данных. Файл ***Problem-Parameters.h*** содержит определения проблемно-зависимых констант.

3.1.2. Логика работы каркаса

Кратко опишем общую логику работы каркаса. ***Головная функция main*** вызывает функцию *BC_MpiRun*, которая выполняет инициализацию MPI и определяет значения переменных:

– *BD_size*: количество запущенных MPI-процессов (не может быть меньше двух);
– *BD_rank*: номер собственного MPI-процесса.

Затем вызывается функция *BC_Init*, которая выделяет необходимую память и определяет значения проблемно-независимых глобальных переменных и структур данных, определенных в файле *BSF-Data.h* (см. стр. 92). Если хотя бы в одном MPI-процессе не удалось выделить необходимую память, процесс-мастер выводит в глобальный поток *cout* диагностическое сообщение «*Error: PC_bsf_Init failed (not enough memory)!*», и все MPI-процессы заканчивают свою работу. Если выделение памяти и инициализация прошли успешно, то процесс с номером *BD_masterRank* выполняет функцию *BC_Master*, а остальные процессы выполняют функцию *BC_Worker*.

Функция BC_Master реализует алгоритм работы мастера. Сначала выводятся параметры задачи с помощью функции *PC_bsf_ParametersOutput*. Затем организуется основной итерационный цикл. В ходе каждой итерации выполняются следующие действия:

1) вызывается функция *BC_MasterMap*, пересылающая всем рабочим в асинхронном режиме (с помощью функции *MPI_Isend*) задание для очередной итерации;
2) вызывается функция *BC_MasterReduce*, получающая от всех рабочих в асинхронном режиме (с помощью функции *MPI_Irecv*) результат выполнения оператора Reduce на подсписках;
3) вызывается предопределенная проблемно-зависимая функция *PC_bsf_ProcessResults*, проверяющая критерий завершения, вычисляющая очередное приближение и формирующая задание для следующей итерации;
4) если определен *#define PP_BSF_ITER_OUTPUT*, выводятся результаты итерации.

Основной итерационный цикл завершается, когда в переменную *BD_exit* функция *PC_bsf_ProcessResults* поместит значение *false*. В этом случае вы-

полняется функция *BC_MasterMap* с параметром *false*, приводящая к завершению процессов-рабочих. После этого происходит вывод результатов с помощью предопределенной проблемно-зависимой функции *PC_bsf_ProblemOutput*, после чего функция *BC_Master* завершает свою работу.

Функция BC_Worker, реализует алгоритм работы рабочего, выполняя в цикле последовательно две функции: *BC_WorkerMap* и *BC_WorkerReduce*. Выход из этого цикла происходит, когда функция *BC_WorkerMap* вернет значение *false*. Функция *BC_WorkerMap* выполняет следующие действия:

1) в синхронном режиме (с помощью функции *MPI_Recv*) получает от мастера задание для выполнения очередной итерации;
2) если поле *exit* в полученном задании содержит значение *true*, происходит выход из функции;
3) применяет предопределенную проблемно-зависимую функцию *PC_bsf_MapF* ко всем элементам своего подсписка «Map», помещая результаты в расширенный список «Reduce».

Функция *BC_WorkerReduce* выполняет следующие действия:

1) выполняет оператор Reduce для своей части расширенного списка «Reduce» с помощью функции *BC_ProcessExtendedReduceList*, которая помещает результат в структуру с указателем *extendedReduceResult_P*;
2) пересылает в синхронном режиме (с помощью функции *MPI_Send*) полученный результат мастеру.

3.2. Визуальный конструктор BSF-программ

В данном разделе описывается процесс проектирования и реализации программной системы BSF-Studio для быстрого создания корректных BSF-про-

грамм на языке C++ с использованием описанного в разделе 3.1 параллельного каркаса. Основными функциональными возможностями BSF-Studio являются следующие:

– пошаговое заполнение проблемно-зависимых частей BSF-каркаса;
– инкрементная (после каждого шага) компиляция программного кода в облачной среде, поддерживающей технологии параллельного программирования OpenMP [119] и MPI [71], с выводом диагностических сообщений компилятора;
– запуск скомпилированного исполняемого кода в облачной среде, поддерживающей технологии параллельного программирования OpenMP и MPI, с выводом результатов выполнения;
– загрузка отлаженного исходного кода.

Программная система BSF-Studio представляет собой веб-приложение с клиент-серверной архитектурой. Клиентская часть представляет собой одностраничное приложение (Single Page Application) [120], серверная часть – Docker-контейнер [195], в котором запускается приложение, выполняющее по запросу компиляцию и запуск программы, и возвращающее результат. Связь между клиентом и севером осуществляется посредством REST API [10].

3.2.1. Проектирование и реализация клиентской части

Интерфейс клиента BSF-Studio представляет собой мастер (wizard) для заполнения BSF-каркаса. Процесс работы мастера разбивается на следующие *шаги*:

1) определение константных параметров модели BSF и параметров задачи (размерность, число уравнений и др.);
2) определение типов данных текущего приближения и элементов списков «Map» и «Reduce»;

3) определение пользовательских переменных и структур данных;
4) реализация пользовательских функций.

Указанные шаги реализуются путем последовательного заполнения HTML-форм, содержащих текстовые поля для заполнения соответствующих фрагментов программного кода.

Реализация интерфейса выполнена с помощью JavaScript-библиотеки React с открытым исходным кодом [13]. React позволяет создавать интерактивные пользовательские интерфейсы на основе компонентного подхода: фрагменты веб-приложения (компоненты) инкапсулируются и используются независимо друг от друга. А за счет большого количества библиотек с открытым исходным кодом, предоставляющих готовые React-компоненты, разработка приложений сводится к подбору, подключению и использованию сторонних модулей.

Для работы с данными использовалась библиотека Redux [13]. Redux берет на себя функции записи, хранения, организации доступа к данным из любого React-компонента. Для реализации HTML-форм использовалась библиотека Redux Form [129]. Redux Form предоставляет готовые компоненты и программную обвязку для реализации HTML-форм и хранения их данных с использованием Redux. Для реализации текстовых полей, предназначенных для работы с программным кодом, использовалась библиотека React Ace [81]. Она предоставляет компонент, в котором реализовано оформление текста в зависимости от языка программирования, выбранного из предоставляемого списка, нумерация строк, настраиваемые панели инструментов.

По нажатию на кнопки «Compile» и «Run» программный код компонуется в файлы и отправляется посредством HTTP-запроса на сервер, где происходит объединение пользовательских файлов и файлов BSF-каркаса и выполняется их компиляция и/или запуск. Результаты компиляции и запуска отображаются в интерфейсе веб-приложения. По нажатию на кнопку «Download» формируются и загружаются файлы BSF-каркаса.

3.2.2. Проектирование и реализация серверной части

Для разработки серверной части приложения BSF-Studio использована технология Docker [108], которая обеспечивает возможность развертывания приложения на базе контейнерной виртуализированной среды. Это позволило создать виртуальное окружение, максимально приближенное к окружению кластерной вычислительной системы. Данный подход позволяет развернуть серверную часть приложения BSF-Studio в рамках любой операционной системы и аппаратного обеспечения, а также упростить процесс развертывания.

Docker предполагает написание сценария, который содержит последовательные инструкции по настройке окружения. В результате выполнения этого сценария строится виртуальный образ, эмулирующий описанное окружение. В рамках этого Docker-сценария необходимо описать на основе какой операционной системы будет функционировать образ, какие подсистемы и библиотеки должны быть установлены, а также произведена настройка и подготовка образа к функционированию приложения внутри него.

Для решения поставленных задач было взято наиболее компактное ядро Linux Alpine [108], содержащее минимальный необходимый набор программ, который позволяет операционной системе функционировать. В качестве компилятора был выбран GNU C Compiler (GCC) [136]. Помимо этого, были установлены библиотеки параллельного программирования OpenMP и MPICH [189]. Для проверки корректности установки и функционирования всех необходимых элементов при сборке образа производится тестовая компиляция нескольких проектов.

Далее необходимо выбрать технологии и средства реализации непосредственно серверного приложения, которое будет осуществлять сборку предоставленного пользователем исходного кода и возвращать результат компиляции посредством REST API.

Серверное приложение разработано на базе программной платформы Node.js, предоставляющей JavaScript-окружение и веб-фреймворка Express.js, который занимается обработкой входящих запросов, и выполняет функции веб-сервера [24]. Также использовались следующие библиотеки:

– Express-request-id – модуль для Express, который позволяет получать параметры из строки запроса [152].
– ShellJS – библиотека, которая позволяет вызывать команды операционной системы [52].
– Multer – библиотека, которая позволяет фреймворку Express принимать и отправлять файлы [159].

Для автоматизации развертывания контейнеризованных приложений клиентской и серверной частей BSF-Studio создан кластер на основе системы Kubernetes [81].

BSF-Studio-API работает согласно следующему алгоритму. Импортируются необходимые библиотеки и модули: Express, express-request-id, ShellJS, Multer. Далее формируются пути до рабочих директорий для текущего запроса и для приложения. Происходит инициализация экземпляра веб-сервера Express и хранилища, в котором будут находиться полученные в текущем запросе файлы и их именование, и инициализация экземпляра Multer. Модуль Express-Request-Id подключается к веб-серверу Express и происходит его запуск. Далее веб-сервер ожидает HTTP-запрос на компиляцию и запуск программного кода. Такой запрос должен содержать файлы проблемно-зависимой части кода для BSF-каркаса, а также имя модели (BSF-MR, BSF-M) в качестве параметров. При получении запроса происходит его обработка и загрузка присланных файлов и файлов каркаса в рабочую директорию. Далее выполняется компиляция, результаты отправляются клиенту в формате JSON, рабочая директория удаляется.

Исходные тексты BSF-Studio свободно доступны в сети Интернет:

– серверная часть: https://github.com/ezhova-nadezhda/BSF-Studio-API;

– клиентская часть: https://github.com/ezhova-nadezhda/BSF-Studio.

3.3. Выводы по главе 3

В третьей главе была описана программная поддержка модели BSF. Она включает в себя параллельный BSF-каркас и веб-приложение BSF-Studio. Параллельный BSF-каркас представляет собой совокупность файлов исходного кода на языке C++, используемых для быстрого создания BSF-программ. BSF-каркас содержит реализацию всех проблемно-независимых функций, организующих параллельное выполнение программы с использованием библиотек MPI и OpenMP. Он также содержит определения (заголовки) проблемно-зависимых функций, реализуемых пользователем. Отличительной особенностью BSF-каркаса является то, что он предполагает поэтапное заполнение пользовательской части исходных кодов и при этом компилируется на всех этапах. Веб-приложение BSF-Studio представляет собой визуальный конструктор BSF-программ. BSF-Studio обеспечивает поэтапное заполнение проблемно-зависимых частей программного кода, компиляцию и запуск приложения в облачной среде. Результаты, приведенные в этой главе, опубликованы в статье [179].

ГЛАВА 4. ВЕРИФИКАЦИЯ МОДЕЛИ BSF

В данной главе представлены результаты по верификации модели BSF. В разделе 4.1 описывается верификация путем имитационного моделирования работы вычислительно-сложного итерационного алгоритма на кластерной вычислительной системе с использованием библиотеки MPI. В разделе 4.2 проверка адекватности модели BSF выполняется на основе итерационного алгоритма Якоби для систем линейных алгебраических уравнений (СЛАУ). В разделе 4.3 исследуется адекватность модели BSF путем ее применения для решения гравитационной задачи.

4.1. Имитационное моделирование BSF-алгоритма

Чтобы проверить адекватность стоимостной метрики модели BSF, был разработан и реализован эмулятор BSF-программ на языке C++ с использованием библиотеки MPI. Исходный код эмулятора свободно доступен на по адресу https://github.com/nadezhda-ezhova/BSF-simulator. Эмулятор использует схему программирования SPMD (Single Program – Multiple Data) [44,142] и включает в себя два вида секций: *секции мастера*, и *секции рабочего*. Секции мастера выполняются, если функция *MPI_Comm_rank* возвращает ноль. *Секция рабочего* выполняется, если функция *MPI_Comm_rank* возвращает значение больше нуля. Задание представляет собой символьную строку фиксированной длины, передаваемую от мастера к рабочим командой *MPI_Isend*. Рабочий считывает данные командой *MPI_Irecv*. Обработка задачи моделируется путем вызова функции *usleep*(t_v), которая приводит к приостановке рабочего MPI-процесса до выполнения указанного количества микросекунд в реальном времени. После обработки мастер и все рабочие выполняют глобальную барьерную синхронизацию с помощью команды *MPI_Barrier*. Затем рабочие посылают мастеру символьную строку

Табл. 5. Значения стоимостных параметров модели BSF.

Параметр	**Семантика**	**Значение (сек.)**
t_r	Время, необходимое для передачи результата мастеру от рабочего (без учета латентности)	0.01
t_p	Время обработки мастером результатов, полученных от рабочих	4.99
t_w	Время выполнения задания бригадой из одного рабочего	500
L	Затраты на инициализацию операции передачи сообщения (латентность)	$2 \cdot 10^{-5}$

определенной длины с использованием команды *MPI_Send*. Мастер считывает результаты командой *MPI_Recv*. Синхронизация выполняется командой *MPI_Waitall*. Далее мастер моделирует обработку результатов, вызывая функцию *usleep(*t_p*)*, которая приводит к приостановке мастер-MPI-процесса до выполнения указанного количества микросекунд в реальном времени. Для верификации формулы вычисления ускорения был введен параметр $v = \lg(t_w / t_s)$, связывающий время t_w, необходимое для выполнения задания бригадой из одного рабочего, и время t_s, необходимое для посылки задания рабочему (без учета латентности). Исследовались следующие значения параметра v: 4, 4.5 и 6. Кривые ускорения, полученные аналитически путем использования формулы ускорения модели BSF, сравнивались с кривыми, полученными в результате численных экспериментов, выполненных с помощью эмулятора BSF-программ. Используемые значения параметров стоимостной метрики модели BSF приведены в табл. 5. Значение латентности L было получено экспериментально как время передачи сообщения в один байт с помощью функций *MPI_Send* и *MPI_Recv*. Значение t_r соответствует пересылке сообщения длиной 14 Мб. Значения параметра t_s зависят от v и t_w. Эта зависимость определяется формулой $t_s = 10^{-v} t_w$. Соответствующие значения параметра t_s были получены варьированием длины задания (см. табл. 6).

Табл. 6. Параметры экспериментов.

v	t_s **(сек.)**	**Объем данных**
4	0.0206978	60 MB
4.5	0.0158160	6 MB
6	0.00048	200 KB

С помощью эмулятора верифицировалась следующая обобщенная формула ускорения модели BSF [174]:

$$a(K) = \frac{2L + 10^{-v} t_w + t_r + t_p + t_w}{K(2L + 10^{-v} t_w) + t_r + t_p + t_w / K}. \qquad (33)$$

Все численные эксперименты проводились на суперкомпьютере «Торнадо-ЮУрГУ» [93]. Результаты верификации формулы (33), представленные на рис. 26, показывают, что стоимостная метрика модели BSF достаточно хорошо предсказывает экспериментальные результаты, полученные с помощью эмулятора BSF-программ. При этом точность аналитических оценок ускорения возрастает с увеличением значения параметра v.

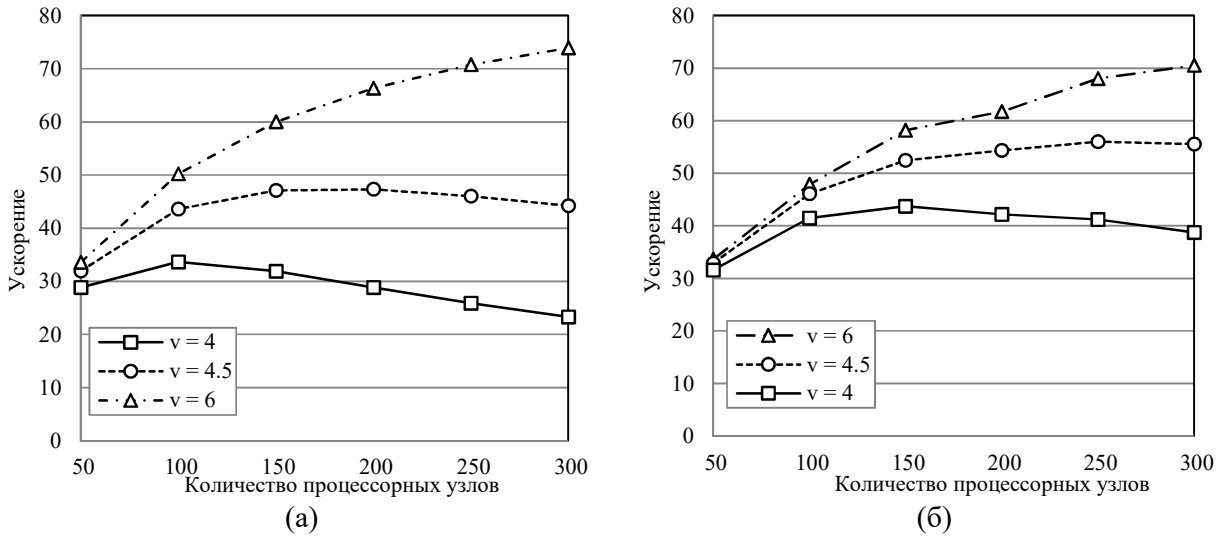

Рис. 26. Верификация формулы для вычисления ускорения:
(а) графики, полученные на основе формулы (33);
(б) графики, полученные экспериментально.

4.2. Метод Якоби для СЛАУ

4.2.1. Построение алгоритма Jacobi-BSF

Итерационный метод Якоби [137] решения СЛАУ был впервые описан немецким математиком Карлом Густавом Якоби в работе [83]. Практическое применение метода Якоби стало возможным в связи с появлением ЭВМ [65]. Научный интерес к методу Якоби не утрачен до сих пор (см., например, [1,167]). Дадим краткое описание метода Якоби. Пусть в евклидовом пространстве $\mathbb{R}^n$ задана совместная система линейных неравенств в матричном виде:

$$Ax = b, \tag{34}$$

где

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}; \quad x = (x_1,\ldots,x_n); \quad b = (b_1,\ldots,b_n).$$

Предполагается, что $a_{ii} \neq 0$, для всех $i = 1,\ldots,n$. Определим матрицу

$$C = \begin{pmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{pmatrix}$$

следующим образом:

$$c_{ij} = \begin{cases} -\dfrac{a_{ij}}{a_{ii}}, \forall j \neq i; \\ 0, \forall j = i. \end{cases}.$$

Определим вектор $d = (d_1,\ldots,d_n)$: $d_i = b_i / a_{ii}$.

Метод Якоби нахождения приближенного решения системы (34) состоит из следующих шагов:

Шаг 1. $k := 0$; $x^{(0)} := d$.

Шаг 2. $x^{(k+1)} := Cx^{(k)} + d$.

Шаг 3. Если $\left\|x^{(k+1)} - x^{(k)}\right\|^2 < \varepsilon$, перейти на шаг 5.

Шаг 4. $k := k+1$; перейти на шаг 2.

Шаг 5. Стоп.

В качестве начального приближения $x^{(0)}$ в методе Якоби может быть взят произвольный вектор в $\mathbb{R}^n$. На шаге 1 в качестве начального приближения $X^{(0)}$ берется вектор d. На шаге 3 в критерии завершения используется евклидова норма $\|\cdot\|$. Достаточным условием сходимости метода Якоби является наличие у матрицы A диагонального преобладания, то есть:

$$|a_{ii}| \geq \left(\sum_{j=1}^{n} |a_{ij}|\right) - |a_{ii}|$$

для всех $i = 1,\ldots,n$, причем хотя бы одно неравенство является строгим. Матрицы с диагональным преобладанием довольно часто возникают в приложениях. В этом случае система (34) имеет единственное решение при любых правых частях.

В соответствии с требованиями модели BSF представим алгоритм Якоби в виде операций над списками с использованием функций высшего порядка *Map* и *Reduce* и назовем такой алгоритм *Jacobi-BSF*. Определим параметризованную функцию $F_x : \{1,\ldots,n\} \to \mathbb{R}^n$ для оператора Map следующим образом:

$$F_x(j) = (c_{1j}x_j,\ldots,c_{nj}x_j) = x_j \begin{pmatrix} c_{1j} \\ \vdots \\ c_{nj} \end{pmatrix}^T . \qquad (35)$$

С неформальной точки зрения функция $F_x(j)$ умножает j-тый столбец матрицы C на j-тую координату вектора x.

Алгоритм 4. Последовательный Jacobi-BSF

1: **input** C, d; $x^{(0)} := d$
2: $k := 0$
3: $[g^1, \dots, g^n] := Map\left(F_{x^{(k)}}, [1, \dots, n]\right)$
4: $g := Reduce(+, [g^1, \dots, g^n])$
5: $x^{(k+1)} := g + d$
6: $k := k + 1$
7: **if** $\left\|x^{(k)} - x^{(k-1)}\right\|^2 < \varepsilon^2$ **goto** 9
8: **goto** 2
9: **output** $x^{(i)}$
10: **stop**

Рис. 27. Последовательный алгоритм Jacobi-BSF.

Алгоритм Jacobi-BSF строится на основе шаблона итерационного алгоритма 2 (рис. 20) и приведен на рис. 27. В данном случае в качестве счетчика итераций фигурирует переменная k. Список «Map» представляет собой список целых чисел $[1,\dots,n]$, где n – размерность пространства. На шаге 3 этот список с помощью функции Map преобразуется в список векторов $[g^1,\dots,g^n]$. На шаге 4 происходит сложение этих векторов с помощью функции Reduce: $g = \sum_{j=1}^{n} g^i$. Итерационный процесс завершается, когда расстояние между соседними приближения станет меньше ε. Параллельный алгоритм строится автоматически на основе шаблона параллельной реализации итерационного алгоритма 3 (рис. 21 на стр. 78).

4.2.2. Аналитическое исследование алгоритма Jacobi-BSF

В рамках одной итерации введем следующие обозначения для анализа масштабируемости алгоритма Jacobi-BSF:

c_s – количество вещественных чисел, передаваемых от мастера рабочему;

c_{Map} – количество арифметических операций, выполняемых на шаге Map (шаг 3 алгоритма);

c_a – количество арифметических операций, необходимое для сложения двух векторов;

c_r – количество вещественных чисел, передаваемых от рабочего мастеру;

c_p – количество арифметических операций, выполняемых мастером на шагах 5 и 7 алгоритма.

Вычислим указанные значения. В начале каждой итерации мастер передает каждому рабочему очередное приближение $x^{(k)}$, являющееся вектором размерности n. Следовательно:

$$c_s = n. \tag{36}$$

На шаге *Map* происходит умножение столбцов матрицы C на соответствующую координату вектора $x^{(k)}$. Для этого требуется n^2 арифметических операций. Следовательно:

$$c_{Map} = n^2. \tag{37}$$

Для сложения двух векторов размерности n требуется n арифметических операций. Следовательно:

$$c_a = n. \tag{38}$$

Вектор, полученный каждым рабочим на шаге *Reduce*, пересылается мастеру. Значит:

$$c_r = n. \tag{39}$$

Выполнение шага 5 требует n арифметических операций. Выполнение шага 7 требует $3n-1$ арифметических операций и одну операцию сравнения. Отсюда получаем следующую формулу:

$$c_p = 4n. \tag{40}$$

Обозначим с помощью τ_{op} время, затрачиваемое рабочим на выполнение одной арифметической операции, и τ_{tr} – время, затрачиваемое на пересылку по сети одного вещественного числа без учета латентности. Тогда, используя

метрику, введенную в разделе 2.3, мы получаем следующие значения для стоимостных параметров алгоритма Jacobi-BSF:

$$t_s = c_s\tau_{tr} = n\tau_{tr}; \tag{41}$$

$$t_{Map} = c_{Map}\tau_{op} = n^2\tau_{op}; \tag{42}$$

$$t_a = c_a\tau_{op} = n\tau_{op}; \tag{43}$$

$$t_r = \tau_{tr}n; \tag{44}$$

$$t_p = c_p\tau_{op} = 4n\tau_{op}. \tag{45}$$

Подставляя в формулу (28) значения правых частей из формул (41) – (45) получаем следующую формулу для оценки ускорения алгоритма Jacobi-BSF:

$$a_{Jacobi-BSF}(K) = \frac{2(L + n\tau_{tr}) + 2n\tau_{op}\left(n+2\right)}{K\left(2(L + n\tau_{tr}) + n\tau_{op}\right) + n\tau_{op}\left(2n/K + 3\right)}. \tag{46}$$

Граница масштабируемости алгоритма Jacobi-BSF получается путем подстановки значений правых частей из формул (41) – (45) в формулу (32):

$$P_{Jacobi-BSF} = \sqrt{\frac{2n^2\tau_{op}}{2(L + \tau_{tr}n) + n\tau_{op}}}. \tag{47}$$

Упростим формулу (47). При $n \to \infty$ имеем

$$2(L + \tau_{tr}n) + \tau_{op}n \approx O(n). \tag{48}$$

Очевидно

$$2n^2\tau_{op} = O(n^2). \tag{49}$$

Подставив правые части уравнений (48) и (49) в (47), получим

$$P_{Jacobi-BSF} \approx \sqrt{\frac{O(n^2)}{O(n)}},$$

что равносильно

$$P_{Jacobi-BSF} \approx \sqrt{O(n)}. \tag{50}$$

Таким образом, граница масштабируемости алгоритма Jacobi-BSF растет пропорционально корню квадратному из размерности задачи n.

4.2.3. Вычислительные эксперименты

Для верификации результатов, полученных аналитическим путем, была выполнена параллельная реализация алгоритма Jacobi-BSF на языке C++ с использованием параллельного BSF-каркаса. Данная реализация свободно доступна в сети Интернет по адресу https://github.com/nadezhda-ezhova/Jacobi-BSF. Для проведения экспериментов была использована масштабируемая система линейных уравнений (34), имеющая следующие матрицу коэффициентов A и столбец свободных членов b:

$$A=\begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & 2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 1 & \cdots & 1 & n \end{pmatrix}, \quad b=\begin{pmatrix} n \\ n+1 \\ \vdots \\ 2n-1 \end{pmatrix}.$$

Исследовано ускорение параллельной реализации алгоритма Jacobi-BSF на суперкомпьютере «Торнадо ЮУрГУ» [93]. Тестирование проводилось для размерностей 1500, 5000, 10000 и 16000. Одновременно для этих же размерностей построены графики ускорения с использованием формулы (46). Для этого экспериментальным путем были определены величины в секундах: $L=1.5\cdot10^{-5}$, $\tau_{op}=2.9\cdot10^{-8}$ и $\tau_{tr}=1.9\cdot10^{-7}$. Результаты приведены на рисунках 28-31. Во всех случаях аналитические оценки оказались очень близки к экспериментальным. Кроме того, границы масштабируемости алгоритма Jacobi-BSF, полученные аналитически, оказались очень близки к границам масштабируемости, полученным в результате вычислительных экспериментов. Это подтверждает адекватность модели параллельных вычислений BSF.

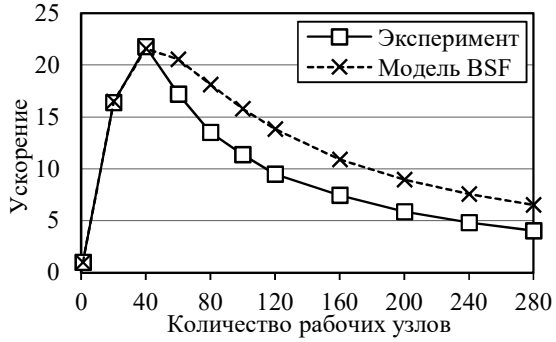

Рис. 28. Jacobi-BSF для n = 1500.

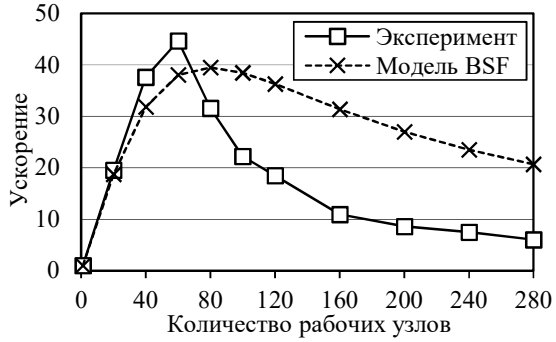

Рис. 29. Jacobi-BSF для n = 5000.

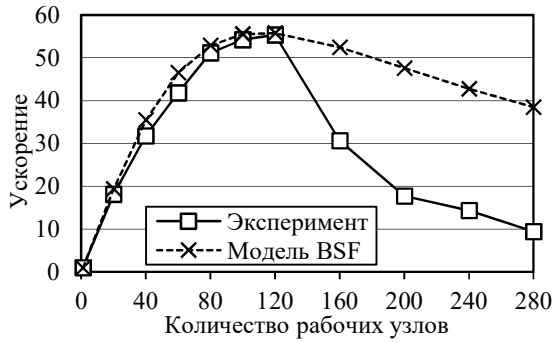

Рис. 30. Jacobi-BSF для n = 10000.

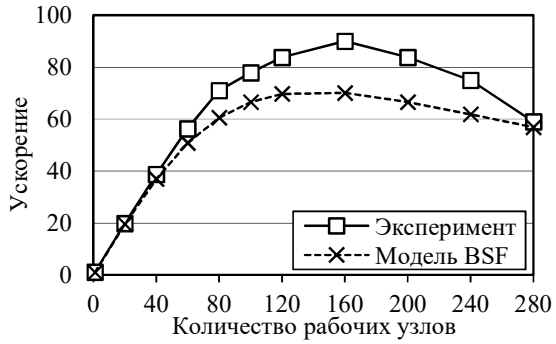

Рис. 31. Jacobi-BSF для n = 16000.

4.3. Гравитационная задача

Выполним верификацию модели BSF с помощью гравитационной задачи, моделирующей движение тела малой массы среди *n* неподвижных тел большой массы. Указанная задача является упрощенным вариантом известной задачи *n* тел [47,106].

4.3.1. Алгоритм Gravitation-BSF

Пусть в $\mathbb{R}^3$ задано конечное множество точек $\mathbb{Y}$, представляющих собой неподвижные тела большой массы. Будем обозначать эти тела следующим образом: $\mathbb{Y} = \{Y_1,\ldots,Y_n\} \subset \mathbb{R}^3$, а их массы соответственно – $\{m_1,\ldots,m_n\}$, где $n = |\mathbb{Y}|$. Пусть также задано некоторое движущееся тело x малой массы m_x. Предполагается, что на x не действуют никакие силы, кроме гравитационных. Для тела x задано его начальное положение $X^{(t_0)} \in \mathbb{R}^3$ и вектор скорости $V^{(t_0)} \in \mathbb{R}^3$ в начальный момент времени t_0. Необходимо рассчитать траекторию движения тела x среди тел $\mathbb{Y}$. Для этого будем последовательно считать новые положения тела x через равные промежутки времени Δt:

$$X^{(t_0)}, X^{(t_0+\Delta t)}, X^{(t_0+2\Delta t)}, X^{(t_0+3\Delta t)}, \ldots \tag{51}$$

По закону всемирного тяготения сила притяжения F_i тела x к телу Y_i $(i = 1,\ldots,n)$ вычисляется по формуле

$$F_i = G\frac{m_i m_x}{\|Y_i - X\|^3}(Y_i - X), \tag{52}$$

где $X \in \mathbb{R}^3$ задает текущие координаты точки x. По второму закону Ньютона ускорение α_i тела x под действием силы F_i вычисляется по формуле

$$\alpha_i = \frac{F_i}{m_x}. \tag{53}$$

Ускорение под действием всех сил $F_1,\ldots,F_n$ вычисляется по формуле

$$\alpha = \sum_{i=1}^{n} \alpha_i. \tag{54}$$

В соответствии с этим вектора скорости для последовательности (51) могут быть посчитаны по следующей итерационной формуле

$$V^{(t+\Delta t)} = V^{(t)} + \alpha^{(t+\Delta t)} \Delta t, \tag{55}$$

где

$$\alpha^{(t+\Delta t)} = \sum_{i=1}^{n} G \frac{m_i}{\left\| Y_i - X^{(t)} \right\|^3} (Y_i - X^{(t)}). \tag{56}$$

Используя (55) получаем

$$X^{(t+\Delta t)} = X^{(t)} + V^{(t+\Delta t)} \Delta t. \tag{57}$$

В контексте гравитационного алгоритма определим список исходных данных A следующим образом:

$$A = [(Y_1, m_1), \ldots, (Y_n, m_n)], \tag{58}$$

то есть A – список пар вида (Y_i, m_i), задающих координаты и массу *i*-того неподвижного тела большой массы. Для произвольной точки $X \in \mathbb{R}^3$ определим функцию $f_X : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}^3$:

$$f_X(Y_i, m_i) = G \frac{m_i}{\left\| Y_i - X \right\|^3} (Y_i - X) \tag{59}$$

для всех $i \in \{1, \ldots, n\}$. Иначе говоря, функция $f_X(Y_i, m_i)$ с использованием формулы (52) вычисляет значение F_i / m_x, где F_i – гравитационная сила, с которой тело Y_i действует на материальную точку с координатами X и массой m_x. Для произвольного $X \in \mathbb{R}^3$ определим список $B \subset \mathbb{R}^3$ следующим образом:

$$B^{(X)} = [f_X(Y_1, m_1), \ldots, f_X(Y_n, m_n)], \tag{60}$$

Алгоритм 5. Алгоритм Gravitation-BSF

1: **input** $A, X^{(t_0)}, V^{(t_0)}, \Delta t, t_0, T$
2: $t := t_0$
3: $B := Map\left(f_{X^{(t)}}, A\right)$
4: $\alpha := Reduce(+, B)$
5: $V^{(t+\Delta t)} := V^{(t)} + \alpha \cdot \Delta t;\ \ X^{(t+\Delta t)} := X^{(t)} + V^{(t+\Delta t)}\Delta t$
6: $t := t + \Delta t$
7: **if** $t \geq T$ **goto** 10
8: **goto** 3
9: **output** $X^{(t)}$
10: **stop**

Рис. 32. Алгоритм Gravitation-BSF.

то есть список B получается из списка A путем применения к нему функции высшего порядка *Map*, использующей в качестве параметра функцию f_X:

$$B = Map(f_X, A).$$

Тогда суммарное ускорение α, получаемое материальной точкой с массой m_x и координатами X под воздействием тел $\mathbb{Y}$, может быть вычислено путем применения к списку B функции высшего порядка *Reduce*, использующей в качестве параметра операцию сложения векторов +:

$$\alpha = Reduce(+, B). \qquad (61)$$

На основе шаблона итерационного алгоритма, представленного на рис. 20 (стр. 77), мы можем записать гравитационный алгоритм Gravitation-BSF в виде операций над списками (см. рис. 32). Здесь t_0 обозначает начальный момент времени, T – конечный момент времени, Δt – шаг по времени. Параллельный алгоритм строится автоматически на основе шаблона параллельной реализации итерационного алгоритма 3 (рис. 21 на стр. 78).

4.3.2. Аналитическое исследование алгоритма Gravitation-BSF

Для простоты мы будем везде далее предполагать, что количество тел большой массы n кратно количеству рабочих K, то есть

$$n = qK \tag{62}$$

при некотором $q \in \mathbb{N}$. В рамках одной итерации ведем следующие обозначения для анализа масштабируемости алгоритма Gravitation-BSF:

c_s – количество вещественных чисел, передаваемых от мастера рабочему;

c_{Map} – количество арифметических операций, выполняемых на шаге *Map* при обработке всего списка исходных данных A (шаг 3 алгоритма 3);

c_a – количество арифметических операций, необходимое для сложения двух векторов в трехмерном пространстве;

c_r – количество вещественных чисел, передаваемых от рабочего мастеру;

c_p – количество арифметических операций, выполняемых мастером на шагах 10 – 12 алгоритма 4.

Вычислим указанные значения. В начале очередной итерации мастер на шаге передает каждому рабочему текущие координаты $X^{(t)}$ тела x, что в совокупности составляет 3 вещественных числа. Следовательно:

$$c_s = 3. \tag{63}$$

На шаге *Map* функция f_X, задаваемая формулой (59), применяется ко всем элементам списка A, имеющего длину n. Предположим, что квадратный корень вычисляется с помощью первых четырех членов ряда Тейлора, что составляет 11 арифметических операций. Тогда однократное вычисление функции f_X потребует 20 арифметических операций. Следовательно, общее количество операции для обработки всего списка A составит

$$c_{Map} = 20n. \tag{64}$$

Для сложения двух векторов размерности 3 требуется 3 арифметические операции. Следовательно:

$$c_a = 3. \tag{65}$$

Трехмерный вектор, полученный рабочим на шаге *Reduce*, пересылается мастеру. Значит:

$$c_r = 3. \tag{66}$$

Выполнение шага 10 алгоритма 5 требует 12 арифметических операций, шага 11 – одну арифметическую операцию и шага 12 – одну операцию сравнения. Отсюда получаем следующую формулу:

$$c_p = 14. \tag{67}$$

Пусть τ_{op} обозначает время, затрачиваемое рабочим на выполнение одной арифметической операции (или операции сравнения), а τ_{tr} – время, затрачиваемое на пересылку по сети одного вещественного числа без учета латентности. Тогда мы получаем следующие значения для стоимостных параметров алгоритма Gravitation-BSF:

$$t_s = c_s\tau_{tr} = 3\tau_{tr}; \tag{68}$$

$$t_{Map} = c_{Map}\tau_{op} = 20n\tau_{op}; \tag{69}$$

$$t_r = c_r\tau_{tr} = 3\tau_{tr}; \tag{70}$$

$$t_a = c_a\tau_{op} = 3\tau_{op}; \tag{71}$$

$$t_p = c_p\tau_{op} = 14\tau_{op}. \tag{72}$$

Подставляя в формулу (28) значения правых частей из формул (68) – (72) получаем следующую формулу для оценки ускорения алгоритма Gravitation-BSF:

$$a_{Gravitation-BSF}(K) = \frac{2L + 6\tau_{tr} + 14\tau_{op} + 20n\tau_{op} + 3l\tau_{op}}{K\left(2L + 6\tau_{tr} + 14\tau_{op}\right) + \dfrac{20n\tau_{op} + 3l\tau_{op}}{K} + 11\tau_{op}}. \tag{73}$$

Граница масштабируемости алгоритма Gravitation-BSF получается путем подстановки значений правых частей из формул (68) – (72) в формулу (32):

$$K_{Gravitation-BSF} = \sqrt{\frac{20n\tau_{op} + 3l\tau_{op}}{2L + 6\tau_{tr} + 3\tau_{op}}}. \quad (74)$$

При $n \to \infty$ имеем

$$K_{Gravitation-BSF} \approx O\left(\sqrt{n}\right). \quad (75)$$

Таким образом, граница масштабируемости алгоритма Gravitation-BSF растет пропорционально корню квадратному из числа, задающего количество неподвижных тел большой массы.

4.3.3. Вычислительные эксперименты

Для верификации результатов, полученных аналитическим путем, была выполнена реализация алгоритма Gravitation-BSF на языке C++ с использованием программного каркаса BSF и библиотеки параллельного программирования MPI. Данная реализация свободно доступна в сети Интернет по адресу https://github.com/nadezhda-ezhova/Gravitation-BSF. С помощью этой реализации исследованы ускорение алгоритма Gravitation-BSF на суперкомпьютере «Торнадо ЮУрГУ» [93]. Тестирование проводилось для 450, 600, 900 и 1200 тел. Выполнялось 10 итераций для каждого случая. Для верификации модели BSF были также построены графики ускорения алгоритма Gravitation-BSF с использованием формулы (73). Для этого экспериментальным путем были определены следующие величины в секундах: $L = 1.5 \cdot 10^{-5}$, $\tau_{op} = 2.9 \cdot 10^{-8}$ и $\tau_{tr} = 1.9 \cdot 10^{-7}$. Результаты приведены на рисунках 33-36. Во всех случаях аналитические оценки оказались очень близки к экспериментальным. Кроме того, границы масштабируемости алгоритма Gravitation-BSF, полученные аналитически, оказались очень близки к границам масштабируемости, полученным в результате вычислительных экспериментов. Это подтверждает адекватность модели параллельных вычислений BSF.

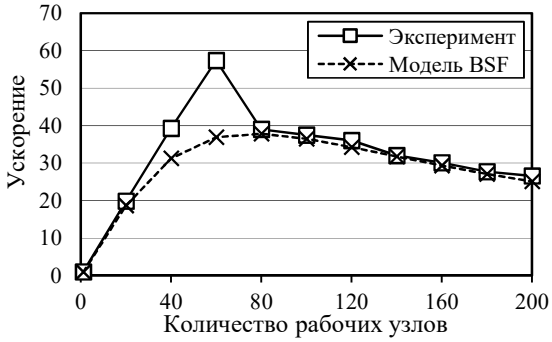

Рис. 33. Эксперимент для $n = 450$.

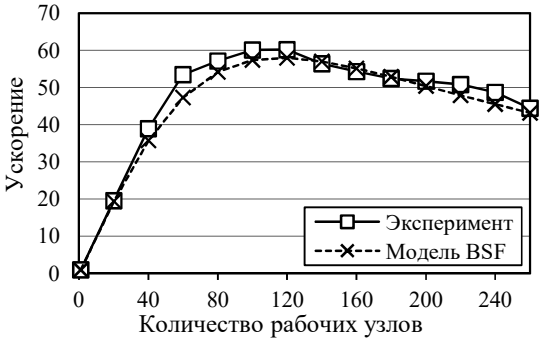

Рис. 35. Эксперимент для $n = 600$.

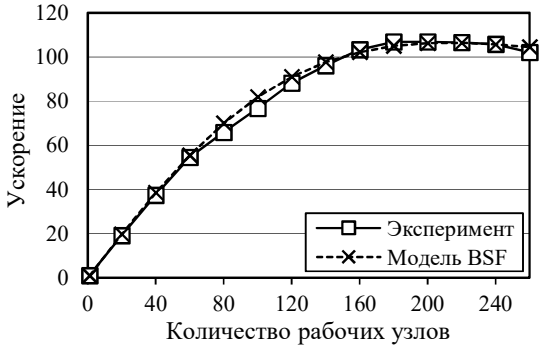

Рис. 34. Эксперимент для $n = 900$.

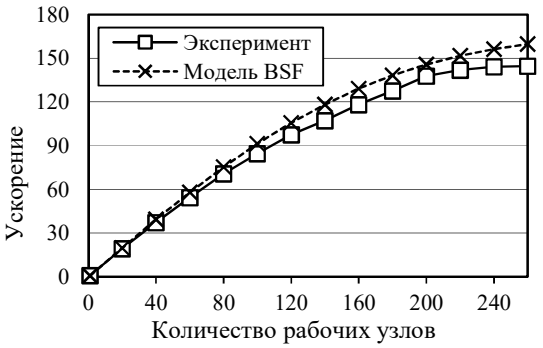

Рис. 36. Эксперимент для $n = 1200$.

4.4. Выводы по главе 4

В этой главе была выполнена верификация модели параллельных вычислений BSF тремя различными способами. Первый способ заключался в разработке эмулятора BSF-программ. С использованием эмулятора были проведены вычислительные эксперименты, на основе которых построены графики ускорений для различных параметров моделируемой BSF-программы. Аналогичные графики построены аналитически с помощью стоимостной метрики модели BSF. Экспериментальные и аналитические результаты оказались очень близки. Второй способ заключался в разработке и реализации BSF-алгоритма на основе итерационного метода Якоби для решения СЛАУ. Указанная реализация была выполнена на языке C++ с использованием параллельного BSF-каркаса. С использованием этой реализации были проведены вычислительные эксперименты, на основе которых построены графики

ускорений для СЛАУ различной размерности. Аналогичные графики были построены аналитически с помощью стоимостной метрики модели BSF. Экспериментальные и аналитические результаты и в этом случае оказались очень близки. При этом модель BSF с высокой точностью предсказывала границу масштабируемости в каждом случае. По этой же схеме происходила верификация третьим способом за тем отличием, что решалась гравитационная задача, являющаяся упрощенным вариантом известной задачи *n* тел, которая хорошо известна в математической физике. Результаты, полученные аналитически и экспериментально, также оказались очень близки, что подтвердило адекватность модели BSF. Кроме этого, следует отметить, что модель BSF применялась для разработки и исследования и других численных алгоритмов (см., например, [147,148]). Полученные результаты позволяют сделать вывод, что модель BSF является адекватной и применима к большому классу задач, допускающих представление алгоритма в виде операций над списками и обладающих высокой вычислительной сложностью. Модель BSF является простой в использовании, а созданный на ее основе параллельный BSF-каркас позволяет быстро разрабатывать эффективные параллельные BSF-программы. Результаты этой главы опубликованы в работах [50,51,175,177].

ЗАКЛЮЧЕНИЕ

В диссертационной работе были рассмотрены вопросы разработки и исследования новой модели параллельных вычислений для итерационных алгоритмов с высокой вычислительной сложностью для многопроцессорных систем с распределенной памятью. Основным научным результатом является создание модели BSF, позволяющей предсказать границу масштабируемости алгоритма на ранних стадиях его проектирования. Разработан компилируемый BSF-каркас на языке C++ с использованием библиотек параллельного программирования MPI и OpenMP, инкапсулирующий все аспекты, связанные с параллелизмом, и позволяющий быстро создавать параллельные реализации алгоритмов, представленных в виде операций над списками. Выполнены проектирование и реализация визуального конструктора программ BSF-Studio, позволяющего автоматизировать процесс создания BSF-программ на языке C++. С использованием BSF-каркаса созданы BSF-программы для известных численных итерационных алгоритмов, на базе которых выполнена верификация BSF-модели.

Основные результаты, полученные в ходе выполнения диссертационного исследования, являются новыми и не покрываются ранее опубликованными научными работами других авторов, обзор которых был дан в разделе 1.3. Отметим основные отличия:

1) ни одна из известных моделей параллельных вычислений не дает формулу, аналогичную формуле (32) на стр. 84, позволяющей аналитически оценить границу масштабируемости параллельного алгоритма;
2) областью применения модели BSF являются параллельные итерационные алгоритмы с большой вычислительной сложностью, представимые в виде операций над списками с использованием функций высшего порядка *Map* и *Reduce*; ни одна из известных научных работ по

моделям параллельных вычислений указанную область ранее не исследовала;

3) модель BSF может быть неадекватной для алгоритмов, не являющихся итерационными, так как не учитывает затраты на чтение исходных данных и инициализацию программы;
4) модель BSF не применима для многопроцессорных систем с общей памятью, так как в этом случае значение знаменателя в формуле (32) на стр. 84 становится пренебрежимо мало по сравнению со значением числителя и граница масштабируемости алгоритма уходит в плюс бесконечность;
5) модель BSF не применима к модели распределённых вычислений MapReduce, разработанной компанией Google для параллельной обработки очень больших наборов данных в компьютерных кластерах по следующим причинам:
   - MapReduce требует представления элементов данных в формате <ключ, значение>, модель BSF этого не требует;
   - в задачах, на которые ориентирована модель MapReduce, временные затраты на арифметические операции (параметры t_{Map} и t_a) во многих случаях пренебрежимо малы по сравнению с затратами на пересылку данных (параметры t_s и t_r); в соответствие с этим формула (32) на стр. 84 будет выдавать для границы масштабируемости значение, близкое к нулю, и таким образом утратит свою работоспособность;
   - вычислительный процесс в MapReduce, как правило, не является итерационным.

Разработанная в ходе настоящего диссертационного исследования модель BSF может применяться для оценки границы масштабируемости параллельных итерационных алгоритмов с высокой вычислительной сложностью,

ориентированных на суперкомпьютеры с кластерной архитектурой. Разработанные параллельный BSF-каркас и визуальный конструктор BSF-Studio могут применяться для быстрого создания и отладки эффективных параллельных BSF-программ для кластерных вычислительных систем.

В качестве направления дальнейших исследований можно выделить создание в рамках BSF-Studio интеллектуального отладчика BSF-программ.

Работа выполнена при финансовой поддержке Правительства РФ в соответствии с Постановлением №211 от 16.03.2013 г. (соглашение № 02.A03.21.0011) и Министерства образования и науки РФ (государственное задание 2.7905.2017/8.9).

ЛИТЕРАТУРА

1. Adsuara J.E. et al. Scheduled Relaxation Jacobi method: Improvements and applications // J. Comput. Phys. Academic Press, 2016. Vol. 321. P. 369–413. DOI:10.1016/j.jcp.2016.05.053.
2. Agarwal A., Negahban S., Wainwright M.J. Noisy matrix decomposition via convex relaxation: Optimal rates in high dimensions // Ann. Stat. Institute of Mathematical Statistics, 2012. Vol. 40, no. 2. P. 1171–1197. DOI:10.1214/12-AOS1000.
3. Agarwal A. et al. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor // Scalable Shared Memory Multiprocessors. Proceedings of a workshop held May 26-27, 1990, in Seattle, Wash. / ed. Dubois M., Thakkar S. Boston, MA: Springer, 1992. P. 239–261. DOI:10.1007/978-1-4615-3604-8_13.
4. Aggarwal A. et al. A model for hierarchical memory // Proceedings of the nineteenth annual ACM conference on Theory of computing - STOC ’87. New York, New York, USA: ACM Press, 1987. P. 305–314. DOI:10.1145/28395.28428.
5. Aggarwal A., Chandra A.K., Snir M. On communication latency in PRAM computations // Proceedings of the first annual ACM symposium on Parallel algorithms and architectures - SPAA ’89. New York, New York, USA: ACM Press, 1989. P. 11–21. DOI:10.1145/72935.72937.
6. Aggarwal A., Chandra A.K., Snir M. Hierarchical memory with block transfer // 28th Annual Symposium on Foundations of Computer Science (sfcs 1987). IEEE, 1987. P. 204–216. DOI:10.1109/SFCS.1987.31.
7. Aho A. V., Hopcroft J.E., Ullman J.D. The design and analysis of computer algorithms. London, Amsterdam, Don Mills, Ontario, Sydney: Addison-Wesley, 1974. 470 p.
8. Alexandrov A. et al. LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation // J. Parallel Distrib. Comput. 1997. Vol. 44, no. 1. P. 71–79. DOI:10.1006/jpdc.1997.1346.
9. Alexandrov V. Route to exascale: Novel mathematical methods, scalable algorithms and Computational Science skills // J. Comput. Sci. Elsevier, 2016. Vol. 14. P. 1–4. DOI:10.1016/J.JOCS.2016.04.014.
10. Allamaraju S. RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity. 1st editio. Yahoo Press, 2010. 316 p.
11. Alpern B. et al. The uniform memory hierarchy model of computation // Algorithmica. Springer-Verlag, 1994. Vol. 12, no. 2–3. P. 72–109. DOI:10.1007/BF01185206.
12. Amaris M. et al. A Simple BSP-based Model to Predict Execution Time in GPU Applications // 2015 IEEE 22nd International Conference on High Performance Computing (HiPC). IEEE, 2015. P. 285–294. DOI:10.1109/HiPC.2015.34.

13. Banks A., Porcello E. Learning React Functional Web Development with React and Redux. O'Reilly Media, 2017. 350 p.
14. Bardine A. et al. NUMA Caches // Encyclopedia of Parallel Computing. Boston, MA: Springer US, 2011. P. 1329–1338. DOI:10.1007/978-0-387-09766-4_16.
15. Bilardi G. et al. On the Effectiveness of D-BSP as a Bridging Model of Parallel Computation // Proceedings of the International Conference on Computational Science - ICCS '01. Part II. Lecture Notes in Computer Science, vol. 2074. Berlin, Heidelberg: Springer, 2001. P. 579–588. DOI:10.1007/3-540-45718-6_63.
16. Bilardi G., Pietracaprina A. Models of Computation, Theoretical // Encyclopedia of Parallel Computing. Boston, MA: Springer US, 2011. P. 1150–1158. DOI:10.1007/978-0-387-09766-4_218.
17. Bilardi G., Pietracaprina A., Pucci G. A Quantitative Measure of Portability with Application to Bandwidth-Latency Models for Parallel Computing // Euro-Par'99 Parallel Processing. Euro-Par 1999. Lecture Notes in Computer Science, vol 1685. Springer, Berlin, Heidelberg, 1999. P. 543–551. DOI:10.1007/3-540-48311-X_76.
18. Bird R., Moor O. De. Algebra of programming. Prentice Hall, 1997. 295 p.
19. Bird R.S. Lectures on Constructive Functional Programming // Constructive Methods in Computing Science. NATO ASI Series F: Computer and Systems Sciences, vol. 55 / ed. Broy M. Berlin, Heidlberg: Springer, 1988. P. 151–216.
20. Bisseling R.H. Parallel Scientific Computation: A Structured Approach using BSP and MPI. New York: Oxford University Press, 2004. P. 325.
21. Bosque J.L., Pastor L. A parallel computational model for heterogeneous clusters // IEEE Trans. Parallel Distrib. Syst. 2006. Vol. 17, no. 12. P. 1390–1400. DOI:10.1109/TPDS.2006.165.
22. Bottomley J. Understanding caching // Linux J. 2004. no. 117. P. 58–62.
23. Bouaziz R. et al. Efficient parallel multi-objective optimization for real-time systems software design exploration // Proceedings of the 27th International Symposium on Rapid System Prototyping - RSP'16. 2016. P. 58–64. DOI:10.1145/2990299.2990310.
24. Brown E. Web Development with Node and Express: Leveraging the JavaScript Stack. 1st ed. O'Reilly Media, 2014. 332 p.
25. Cameron K.W., Sun X.-H. Quantifying locality effect in data access delay: memory logP // Proceedings of the 2003 IEEE International Parallel and Distributed Processing Symposium (IPDPS'03). IEEE Comput. Soc, 2003. P. 8. DOI:10.1109/IPDPS.2003.1213137.
26. Cameron K.W., Ge R. Predicting and Evaluating Distributed Communication Performance // Proceedings of the 2004 ACM/IEEE Conference on Supercomputing. IEEE, 2004. P. 15. DOI:10.1109/SC.2004.40.

27. Cameron K.W., Ge R., Sun X.-H. lognP and log3P: Accurate Analytical Models of Point-to-point Communication in Distributed Systems // IEEE Trans. Comput. 2007. Vol. 56, no. 3. P. 314–327. DOI:10.1109/TC.2007.38.
28. Campbell D.K.G. A survey of models of parallel computation. Technical Report No.YCS-97-278. 1997. 37 p.
29. Cantú-Paz E., Goldberg D.E. On the Scalability of Parallel Genetic Algorithms // Evol. Comput. 1999. Vol. 7, no. 4. P. 429–449. DOI:10.1162/evco.1999.7.4.429.
30. Cappello F. et al. HiHCoHP-Toward a realistic communication model for hierarchical hyperclusters of heterogeneous processors // Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001. IEEE Comput. Soc., 2001. P. 6. DOI:10.1109/IPDPS.2001.924978.
31. Cappello F. et al. An algorithmic model for heterogeneous hyper-clusters: rationale and experience // Int. J. Found. Comput. Sci. World Scientific Publishing Company, 2005. Vol. 16, no. 02. P. 195–215. DOI:10.1142/S0129054105002942.
32. Censor Y., Elfving T., Herman G.T. Averaging Strings of Sequential Iterations for Convex Feasibility Problems // Inherently Parallel Algorithms in Feasibility and Optimization and their Applications. Studies in Computational Mathematics, Vol. 8 / ed. Butnariu D., Censor Y., Reich S. Amsterdam, The Netherlands: Elsevier Science Publishers, 2001. P. 101–113. DOI:10.1016/S1570-579X(01)80009-4.
33. Censor Y. Sequential and parallel projection algorithms for feasibility and optimization // Proc. SPIE 4553, Visualization and Optimization Techniques, (25 September 2001) / ed. Censor Y., Ding M. International Society for Optics and Photonics, 2001. P. 1–9. DOI:10.1117/12.441550.
34. Censor Y. Mathematical optimization for the inverse problem of intensity-modulated radiation therapy // Intensity-Modulated Radiation Therapy: The State of the Art / ed. Palta J.R., Mackie T.R. Madison, WI: Medical Physics Publishing, 2003. P. 25–49. DOI:10.1118/1.1628279.
35. Censor Y. et al. New Methods for Linear Inequalities // SIAM J. Sci. Comput. Society for Industrial and Applied Mathematics, 2008. Vol. 30, no. 1. P. 473–504. DOI:10.1137/050639399.
36. Ceze L.H. Shared-Memory Multiprocessors // Encyclopedia of Parallel Computing. Boston, MA, USA: Springer US, 2011. P. 1810–1812. DOI:10.1007/978-0-387-09766-4_142.
37. Chetverushkin B.N. et al. Numerical algorithms for HPC Systems and fault tolerance // Parallel Computational Technologies. PCT 2019. Communications in Computer and Information Science, vol. 1063. 2019. P. 34–44. DOI:10.1007/978-3-030-28163-2_3.
38. Cole M.I. Algorithmic Skeletons: Structured Management of Parallel Computation. Cambridge, MA, USA: MIT Press, 1991. 170 p.

39. Cole R., Zajicek O. The APRAM: incorporating asynchrony into the PRAM model // Proceedings of the first annual ACM symposium on Parallel algorithms and architectures - SPAA ’89. New York, New York, USA: ACM Press, 1989. P. 169–178. DOI:10.1145/72935.72954.
40. Cook S.A., Reckhow R.A. Time bounded random access machines // J. Comput. Syst. Sci. Academic Press, 1973. Vol. 7, no. 4. P. 354–375. DOI:10.1016/S0022-0000(73)80029-7.
41. Cook S., Dwork C., Reischuk R. Upper and Lower Time Bounds for Parallel Random Access Machines without Simultaneous Writes // SIAM J. Comput. Society for Industrial and Applied Mathematics, 1986. Vol. 15, no. 1. P. 87–97. DOI:10.1137/0215006.
42. Culler D. et al. LogP: towards a realistic model of parallel computation // Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPOPP’93. New York, New York, USA: ACM Press, 1993. P. 1–12. DOI:10.1145/155332.155333.
43. Dandamudi S.P. ARM Architecture // Guide to RISC Processors. New York: Springer, 2005. P. 121–145. DOI:10.1007/0-387-27446-4_8.
44. Darema F. et al. A single-program-multiple-data computational model for EPEX/FORTRAN // Parallel Comput. 1988. Vol. 7, no. 1. P. 11–24. DOI:10.1016/0167-8191(88)90094-4.
45. Darema F. SPMD Computational Model // Encyclopedia of Parallel Computing. Boston, MA: Springer US, 2011. P. 1933–1943. DOI:10.1007/978-0-387-09766-4_26.
46. Depolli M., Trobec R., Filipič B. Asynchronous Master-Slave Parallelization of Differential Evolution for Multi-Objective Optimization // Evol. Comput. 2012. Vol. 21, no. 2. P. 1–31. DOI:10.1162/EVCO_a_00076.
47. Diacu F. The solution of the n-body problem // Math. Intell. 1996. Vol. 18, no. 3. P. 66–70. DOI:10.1007/BF03024313.
48. Dongarra J., Gottlieb S., Kramer W.T.C. Race to Exascale // Comput. Sci. Eng. 2019. Vol. 21, no. 1. P. 4–5. DOI:10.1109/MCSE.2018.2882574.
49. Elgot C.C., Robinson A. Random-Access Stored-Program Machines, an Approach to Programming Languages // J. ACM. ACM, 1964. Vol. 11, no. 4. P. 365–399. DOI:10.1145/321239.321240.
50. Ezhova N. Verification of BSF Parallel Computational Model // 3rd Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists (Ural-PDC). CEUR Workshop Proceedings. Vol. 1990. CEUR-WS.org, 2017. P. 30–39.
51. Ezhova N.A., Sokolinsky L.B. Scalability Evaluation of Iterative Algorithms Used for Supercomputer Simulation of Physical processes // Proceedings - 2018 Global Smart Industry Conference, GloSIC 2018. Article number 8570107. IEEE, 2018. 10 p. DOI:10.1109/GloSIC.2018.8570131.

52. Fischer N., Freitag B. ShellJS - Unix shell commands for Node.js (GitHub Repository) [Electronic resource].
53. Forsell M. E - A Language for Thread-Level Parallel Programming on Synchronous Shared Memory NOCs // WSEAS Trans. Comput. 2004. Vol. 3, no. 3. P. 807–812.
54. Forsell M. A PRAM-NUMA Model of Computation for Addressing Low-TLP Workloads // Int. J. Netw. Comput. [Hiroshima University], 2011. Vol. 1, no. 1. P. 21–35.
55. Forsell M. et al. Hardware and Software Support for NUMA Computing on Configurable Emulated Shared Memory Architectures // 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum. IEEE, 2013. P. 640–648. DOI:10.1109/IPDPSW.2013.146.
56. Forsell M., Leppanen V. An Extended PRAM-NUMA Model of Computation for TCF Programming // Int. J. Netw. Comput. 2013. Vol. 3, no. 1. P. 98–115.
57. Fortune S., Wyllie J. Parallelism in random access machines // Proceedings of the tenth annual ACM symposium on Theory of computing - STOC ’78. New York, New York, USA: ACM Press, 1978. P. 114–118. DOI:10.1145/800133.804339.
58. Gerbessiotis A. V. Extending the BSP model for multi-core and out-of-core computing: MBSP // Parallel Comput. Elsevier B.V., 2015. Vol. 41. P. 90–102. DOI:10.1016/j.parco.2014.12.002.
59. Germanov M.A., Spiridonov V.S. On a method of solving systems of non-linear inequalities // USSR Comput. Math. Math. Phys. No longer published by Elsevier, 1966. Vol. 6, no. 2. P. 194–196. DOI:10.1016/0041-5553(66)90066-8.
60. Gibbons J. An Introduction to the Bird-Meertens Formalism // Proceedings of the First New Zealand Formal Program Development Colloquium. (Working paper 94/18) / ed. Reeves S. Hamilton, New Zealand: University of Waikato, Department of Computer Science, 1994. P. 1–12.
61. Gibbons P.B. A more practical PRAM model // Proceedings of the first annual ACM symposium on Parallel algorithms and architectures - SPAA ’89. New York, New York, USA: ACM Press, 1989. P. 158–168. DOI:10.1145/72935.72953.
62. Gibbons P.B., Matias Y. Can a Shared-Memory Model Serve as a Bridging Model for Parallel Computation? // Theory Comput. Syst. 1999. Vol. 32, no. 3. P. 327–359. DOI:10.1007/s002240000121.
63. Gibbons P.B., Matias Y., Ramachandran V. The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms // SIAM J. Comput. 1998. Vol. 28, no. 2. P. 733–769. DOI:10.1137/S009753979427491.
64. Goldschlager L.M. A unified approach to models of synchronous parallel machines // Proceedings of the tenth annual ACM symposium on Theory of

computing - STOC ’78. New York, New York, USA: ACM Press, 1978. P. 89–94. DOI:10.1145/800133.804336.
65. Goldstine H.H., Murray F.J., von Neumann J. The Jacobi Method for Real Symmetric Matrices // J. ACM. ACM, 1959. Vol. 6, no. 1. P. 59–96. DOI:10.1145/320954.320960.
66. Gonzalez-Gutierrez E., Todorov M.I. A relaxation method for solving systems with infinitely many linear inequalities // Optim. Lett. Springer-Verlag, 2012. Vol. 6, no. 2. P. 291–298. DOI:10.1007/s11590-010-0244-4.
67. Gonzalez-Velez H., Leyton M. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers // Softw. Pract. Exp. John Wiley & Sons, Ltd., 2010. Vol. 40, no. 12. P. 1135–1160. DOI:10.1002/spe.1026.
68. Goudreau M. et al. Towards efficiency and portability: Programming with the BSP Model // Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures - SPAA’96. New York, NY, USA: ACM Press, 1996. P. 1–12. DOI:10.1145/237502.237503.
69. Grama A. et al. Architecture Independent Analysis of Parallel Programs // Proceedings of the International Conference on Computational Science - ICCS ’01. Part II. Lecture Notes in Computer Science, vol. 2074. Berlin, Heidelberg: Springer, 2001. P. 599–608. DOI:10.1007/3-540-45718-6_65.
70. Gropp W. et al. A high-performance, portable implementation of the MPI message passing interface standard // Parallel Comput. 1996. Vol. 22, no. 6. P. 789–828. DOI:10.1016/0167-8191(96)00024-5.
71. Gropp W. MPI 3 and Beyond: Why MPI Is Successful and What Challenges It Faces // Recent Advances in the Message Passing Interface. EuroMPI 2012. Lecture Notes in Computer Science, vol. 7490 / ed. Träff J.L., Benkner S., Dongarra J.J. Berlin, Heidelberg: Springer, 2012. P. 1–9. DOI:10.1007/978-3-642-33518-1_1.
72. Gropp W., Lusk E., Skjellum A. Using MPI: portable parallel programming with the message-passing interface. Second Edi. MIT Press, 1999.
73. Gubin L.G., Polyak B.T., Raik E.V. The method of projections for finding the common point of convex sets // USSR Comput. Math. Math. Phys. No longer published by Elsevier, 1967. Vol. 7, no. 6. P. 1–24. DOI:10.1016/0041-5553(67)90113-9.
74. Hadjidimos A. A survey of the iterative methods for the solution of linear systems by extrapolation, relaxation and other techniques // J. Comput. Appl. Math. North-Holland, 1987. Vol. 20. P. 37–51. DOI:10.1016/0377-0427(87)90124-5.
75. Hageman L.A., Young D.M. Applied iterative methods. New York, London, Toronto, Sydney, San Francisco: Academic Press, 1981. 386 p.
76. Hartmanis J. Computational complexity of random access stored program machines // Math. Syst. Theory. Springer-Verlag, 1971. Vol. 5, no. 3. P. 232–245. DOI:10.1007/BF01694180.

77. Hennessy J.L., Patterson D.A. Computer Architecture: A Quantitative Approach // Computer. Fifth Edit. Morgan Kaufmann, 2011. 856 p.
78. Heywood T., Ranka S. A practical hierarchical model of parallel computation I. The model // J. Parallel Distrib. Comput. Academic Press, 1992. Vol. 16, no. 3. P. 212–232. DOI:10.1016/0743-7315(92)90034-K.
79. Hoefler T. et al. LogfP - a model for small messages in InfiniBand // Proceedings 20th IEEE International Parallel & Distributed Processing Symposium. Washington, DC, USA: IEEE Computer Society, 2006. P. 319–319. DOI:10.1109/IPDPS.2006.1639624.
80. Hoefler T., Schneider T., Lumsdaine A. LogGOPSim – Simulating Large-Scale Applications in the LogGOPS Model // Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - HPDC '10. New York, New York, USA: ACM Press, 2010. P. 597–604. DOI:10.1145/1851476.1851564.
81. Hrisho J. React-Ace (GitHub Repository) [Electronic resource].
82. Ino F., Fujimoto N., Hagihara K. LogGPS: A parallel computational model for synchronization analysis // ACM SIGPLAN Not. 2001. Vol. 36, no. 7. P. 133–142. DOI:10.1145/568014.379592.
83. Jacobi C.G.J. Ueber eine neue Auflösungsart der bei der Methode der kleinsten Quadrate vorkommenden lineären Gleichungen // Astron. Nachrichten. Wiley-Blackwell, 1845. Vol. 22, no. 20. P. 297–306. DOI:10.1002/asna.18450222002.
84. JaJa J.F. An Introduction to Parallel Algorithms. Redwood City, CA, USA: Addison Wesley Publishing Co., Reading, 1992. 576 p.
85. JaJa J.F. PRAM (Parallel Random Access Machines) // Encyclopedia of Parallel Computing. Boston, MA: Springer US, 2011. P. 1608–1615. DOI:10.1007/978-0-387-09766-4_23.
86. Jepsen T.C. InfiniBand // Distributed Storage Networks: Architecture, Protocols and Management. Chichester, West Sussex, England: John Wiley & Sons, 2013. P. 159–174. DOI:10.1002/9780470871461.ch6.
87. Kalbe G. The European Approach to the Exascale Challenge // Comput. Sci. Eng. 2019. Vol. 21, no. 1. P. 42–47. DOI:10.1109/MCSE.2018.2884139.
88. Karp R.M., Ramachandran V. Parallel Algorithms for Shared-Memory Machines // Handbook of theoretical computer science. Volume A: Algorithms and Complexity / ed. Van Leeuwen J. Amsterdam, New York, Oxford, Tokyo: Elsevier, 1990. P. 871–941.
89. Kelley C.T. Iterative Methods for Linear and Nonlinear Equations. Philadelphia: Society for Industrial and Applied Mathematics, 1995. xiii + 156 p. DOI:10.1137/1.9781611970944.
90. Kielmann T., Bal H.E., Verstoep K. Fast Measurement of LogP Parameters for Message Passing Platforms // Parallel and Distributed Processing. IPDPS 2000. Lecture Notes in Computer Science, vol 1800. Berlin,

Heidelberg: Springer, 2000. P. 1176–1183. DOI:10.1007/3-540-45591-4_162.
91. Kielmann T., Gorlatch S. Bandwidth-Latency Models (BSP, LogP) // Encyclopedia of Parallel Computing. Boston, MA: Springer US, 2011. P. 107–112. DOI:10.1007/978-0-387-09766-4_189.
92. Konnov I. Combined Relaxation Methods for Variational Inequalities. Lecture Notes in Economics and Mathematical Systems, vol. 495. Berlin, Heidelberg: Springer, 2001. 183 p. DOI:10.1007/978-3-642-56886-2.
93. Kostenetskiy P.S., Safonov A.Y. SUSU Supercomputer Resources // Proceedings of the 10th Annual International Scientific Conference on Parallel Computing Technologies (PCT 2016). CEUR Workshop Proceedings. Vol. 1576. 2016. P. 561–573.
94. Kothe D., Lee S., Qualters I. Exascale Computing in the United States // Comput. Sci. Eng. 2019. Vol. 21, no. 1. P. 17–29. DOI:10.1109/MCSE.2018.2875366.
95. Kubiatowicz J., Agarwal A. Anatomy of a message in the Alewife multiprocessor // ACM International Conference on Supercomputing 25th Anniversary Volume. New York, NY, USA: ACM Press, 2014. P. 193–204. DOI:10.1145/2591635.2667168.
96. de la Torre P., Kruskal C.P. Towards a single model of efficient computation in real parallel machines // Futur. Gener. Comput. Syst. North-Holland, 1992. Vol. 8, no. 4. P. 395–408. DOI:10.1016/0167-739X(92)90071-I.
97. Ladner R.E., Fischer M.J. Parallel Prefix Computation // J. ACM. 1980. Vol. 27, no. 4. P. 831–838. DOI:10.1145/322217.322232.
98. Leasure B. et al. Parallel Skeletons // Encyclopedia of Parallel Computing. Boston, MA: Springer US, 2011. P. 1417–1422. DOI:10.1007/978-0-387-09766-4_24.
99. Leoncini M. Parallel models of computation: an introductory survey // Calcolo. Springer-Verlag, 1989. Vol. 26, no. 2–4. P. 209–236. DOI:10.1007/BF02575730.
100. Leung J.Y.-T., Zhao H. Scheduling problems in master-slave model // Ann. Oper. Res. 2008. Vol. 159, no. 1. P. 215–231. DOI:10.1007/s10479-007-0271-4.
101. De Loera J.A., Haddock J., Needell D. A Sampling Kaczmarz-Motzkin Algorithm for Linear Feasibility // SIAM J. Sci. Comput. Society for Industrial and Applied Mathematics, 2017. Vol. 39, no. 5. P. S66–S87. DOI:10.1137/16M1073807.
102. Lu F., Song J., Pang Y. HLognGP: A parallel computation model for GPU clusters // Concurr. Comput. Pract. Exp. 2015. Vol. 27, no. 17. P. 4880–4896. DOI:10.1002/cpe.3475.
103. Luccio F., Pagli L. A model of sequential computation with Pipelined access to memory // Math. Syst. Theory. Springer-Verlag, 1993. Vol. 26,

no. 4. P. 343–356. DOI:10.1007/BF01189854.
104. Maggs B.M., Matheson L.R., Tarjan R.E. Models of parallel computation: a survey and synthesis // Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences. IEEE Comput. Soc. Press, 1995. P. 61–70. DOI:10.1109/HICSS.1995.375476.
105. Mansour Y., Nisan N., Vishkin U. Trade-offs between Communication Throughput and Parallel Time // J. Complex. Academic Press, 1999. Vol. 15, no. 1. P. 148–166. DOI:10.1006/JCOM.1998.0498.
106. Marciniak A. Numerical Solutions of the N-Body Problem. Dordrecht, Boston, Lancaster: D. Reidel Publishing Company, 1985. 242 p. DOI:10.1007/978-94-009-5412-0.
107. Mathias E.N. et al. DEVOpT: A distributed architecture supporting heuristic and metaheuristic optimization methods // Proc. ACM Symp. Appl. Comput. 2002. P. 870–875. DOI:10.1145/508791.508960.
108. Matthias K., Kane S.P. Docker: Up & Running: Shipping Reliable Containers in Production. 2nd Editio. O'Reilly Media, 2018. 352 p.
109. McColl W.F. Scalable computing // J. van Leeuwen (eds). Computer Science Today: Recent Trends and Developments. Lecture Notes in Computer Science, vol. 1000. Berlin, Heidelberg: Springer, 1995. P. 46–61. DOI:10.1007/BFb0015236.
110. McColl W.F., Tiskin A. Memory-Efficient Matrix Multiplication in the BSP Model // Algorithmica. Springer-Verlag, 1999. Vol. 24, no. 3–4. P. 287–297. DOI:10.1007/PL00008264.
111. Mead C.A., Conway L.A. Introduction to VLSI systems. Boston, MA, USA: Addison-Wesley, 1980. 396 p.
112. Mehlhorn K., Vishkin U. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories // Acta Inform. Springer-Verlag, 1984. Vol. 21, no. 4. P. 339–374. DOI:10.1007/BF00264615.
113. Moritz C.A., Frank M.I. LoGPC: Modeling network contention in message-passing programs // IEEE Trans. Parallel Distrib. Syst. 2001. Vol. 12, no. 4. P. 404–415. DOI:10.1109/71.920589.
114. Moritz C.A., Frank M.I. LoGPC: Modeling Network Contention in Message-Passing Programs // ACM SIGMETRICS Perform. Eval. Rev. New York, New York, USA: ACM Press, 1998. Vol. 26, no. 1. P. 254–263. DOI:10.1145/277851.277933.
115. Nasri W., Tarhouni O., Slimi N. PLP: Towards a realistic and accurate model for communication performances on hierarchical cluster-based systems // 2008 IEEE International Symposium on Parallel and Distributed Processing. IEEE, 2008. P. 1–8. DOI:10.1109/IPDPS.2008.4536486.
116. Nayfeh B.A., Olukotun K. A single-chip multiprocessor // Computer (Long. Beach. Calif). 1997. Vol. 30, no. 9. P. 79–85. DOI:10.1109/2.612253.

117. Owens J.D. et al. GPU Computing // Proc. IEEE. 2008. Vol. 96, no. 5. P. 879–899. DOI:10.1109/JPROC.2008.917757.
118. Palmer J., Steele G.L. Connection Machine model CM-5 system overview // Frontiers '92, the Fourth Symposium on the Frontiers of Massive Parallel Computation, October 19-21, 1992, McLean, Virginia. IEEE Computer Society Press, 1992. P. 474–483. DOI:10.1109/FMPC.1992.234877.
119. Pas R. van der, Stotzer E., Terboven C. Using OpenMP—The Next Step: Affinity, Accelerators, Tasking, and SIMD (Scientific and Engineering Computation). 1st ed. The MIT Press, 2017. 392 p.
120. Patel Y. White Paper On Single Page Application. Knowarth [Electronic resource]. 2015. P. 8. URL: https://www.knowarth.com/wp-content/uploads/2015/02/Single-Page-Application-White-Paper.pdf (accessed: 19.09.2019).
121. Petitet A. et al. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers [Electronic resource]. 2018. URL: https://www.netlib.org/benchmark/hpl/ (accessed: 02.07.2019).
122. Pfister G.F. In Search of Clusters. 2nd Editio. Upper Saddle River, NJ: Prentice Hall, 1998. 575 p.
123. De Pierro A.R., Iusem A.N. A simultaneous projections method for linear inequalities // Linear Algebra Appl. North-Holland, 1985. Vol. 64. P. 243–253. DOI:10.1016/0024-3795(85)90280-0.
124. Pippenger N. On simultaneous resource bounds // 20th Annual Symposium on Foundations of Computer Science (SFCS 1979). San Juan, Puerto Rico: IEEE, 1979. P. 307–311. DOI:10.1109/SFCS.1979.29.
125. Pippenger N. Pebbling with an auxiliary pushdown // J. Comput. Syst. Sci. Academic Press, 1981. Vol. 23, no. 2. P. 151–165. DOI:10.1016/0022-0000(81)90011-8.
126. Qian D., Luan Z. High Performance Computing Development in China: A Brief Review and Perspectives // Comput. Sci. Eng. 2019. Vol. 21, no. 1. P. 6–16. DOI:10.1109/MCSE.2018.2875367.
127. Qiao X., Chen S., Yang L.T. HPM: a hierarchical model for parallel computations // Int. J. High Perform. Comput. Netw. 2004. Vol. 1, no. 1–3. P. 117–127. DOI:10.1504/IJHPCN.2004.007571.
128. Ranade A.G. How to emulate shared memory // J. Comput. Syst. Sci. Academic Press, 1991. Vol. 42, no. 3. P. 307–326. DOI:10.1016/0022-0000(91)90005-P.
129. Rasmussen E. Redux Form: The best way to manage your form state in Redux [Electronic resource].
130. Rico-Gallego J.-A., Díaz-Martín J.-C. τ-Lop: Modeling performance of shared memory MPI // Parallel Comput. North-Holland, 2015. Vol. 46. P. 14–31. DOI:10.1016/J.PARCO.2015.02.006.
131. Rico-Gallego J.-A., Lastovetsky A.L., Diaz-Martin J.-C. Model-Based

Estimation of the Communication Cost of Hybrid Data-Parallel Applications on Heterogeneous Clusters // IEEE Trans. Parallel Distrib. Syst. 2017. Vol. 28, no. 11. P. 3215–3228. DOI:10.1109/TPDS.2017.2715809.
132. Rico-Gallego J.A. et al. A Survey of Communication Performance Models for High-Performance Computing // ACM Comput. Surv. ACM, 2019. Vol. 51, no. 6. P. 1–36. DOI:10.1145/3284358.
133. Robey R.W. Parallel Algorithms for the Exascale Era. Technical Report LA-UR-16-28005. Los Alamos, NM (United States), 2017. 38 p. DOI:10.2172/1330080.
134. Rochange C., Uhrig S., Sainrat P. Memory Hierarchy // Time-Predictable Architectures. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2014. P. 69–104. DOI:10.1002/9781118790229.ch4.
135. Rothenberg J. The nature of modeling // Artificial Intelligence, Simulation, and Modeling / ed. Widman L.E., Loparo K.A., Nielson N. New York, NY, USA: John Wiley and Sons, 1989. P. 75–92.
136. Rothwell T., Youngman J. The GNU C Reference Manual [Electronic resource]. Free Software Foundation, Inc, 2007. P. 86.
137. Rutishauser H. The Jacobi Method for Real Symmetric Matrices // Handbook for Automatic Computation, vol 2. Linear Algebra / ed. Bauer F.L. Berlin, Heidelberg: Springer Berlin Heidelberg, 1971. P. 202–211. DOI:10.1007/978-3-662-39778-7_12.
138. Sahni S. Scheduling master-slave multiprocessor systems // IEEE Trans. Comput. 1996. Vol. 45, no. 10. P. 1195–1199. DOI:10.1109/12.543712.
139. Sahni S., Vairaktarakis G. The master-slave paradigm in parallel computer and industrial settings // J. Glob. Optim. 1996. Vol. 9, no. 3–4. P. 357–377. DOI:10.1007/BF00121679.
140. Shepherdson J.C., Sturgis H.E. Computability of Recursive Functions // J. ACM. ACM, 1963. Vol. 10, no. 2. P. 217–255. DOI:10.1145/321160.321170.
141. Shonkwiler R.W., Lefton L. Iterative Methods for Linear Systems // An Introduction to Parallel and Vector Scientific Computing. Cambridge: Cambridge University Press, 2006. P. 186–205. DOI:10.1017/CBO9780511617935.009.
142. Silva L.M., Buyya R. Parallel programming models and paradigms // High Performance Cluster Computing: Architectures and Systems. Vol. 2. 1999. P. 4–27.
143. Skillicorn D.B. Parallelism and the Bird-Meertens Formalism. Kingston, Canada, 1992. 16 p.
144. Skillicorn D.B., Talia D. Models and languages for parallel computation // ACM Comput. Surv. 1998. Vol. 30, no. 2. P. 123–169. DOI:10.1145/280277.280278.
145. Snir M. Distributed-Memory Multiprocessor // Encyclopedia of Parallel

Computing. Boston,MA: Springer US, 2011. P. 574–578.
146. Snyder L. Type Architectures, Shared Memory, and the Corollary of Modest Potential // Annu. Rev. Comput. Sci. 1986. Vol. 1, no. 1. P. 289–317. DOI:10.1146/annurev.cs.01.060186.001445.
147. Sokolinskaya I.M., Sokolinsky L.B. Scalability Evaluation of Cimmino Algorithm for Solving Linear Inequality Systems on Multiprocessors with Distributed Memory // Supercomput. Front. Innov. 2018. Vol. 5, no. 2. P. 11–22. DOI:10.14529/jsfi180202.
148. Sokolinskaya I., Sokolinsky L.B. Scalability Evaluation of NSLP Algorithm for Solving Non-Stationary Linear Programming Problems on Cluster Computing Systems // Supercomput. RuSCDays 2017. Commun. Comput. Inf. Sci. / ed. Voevodin V., Sobolev S. Cham: Springer, 2017. Vol. 793. P. 40–53. DOI:10.1007/978-3-319-71255-0_4.
149. Sorensen B. Japan's Flagship 2020 "Post-K" System // Comput. Sci. Eng. 2019. Vol. 21, no. 1. P. 48–49. DOI:10.1109/MCSE.2018.2886646.
150. Sterling T. et al. SLOWER: A Performance Model for Exascale Computing // Supercomput. Front. Innov. 2014. Vol. 1, no. 2. P. 42–57. DOI:10.14529/jsfi140203.
151. Sterling T.L. Clusters // Encyclopedia of Parallel Computing. Boston,MA: Springer US, 2011. P. 289–297. DOI:10.1007/978-0-387-09766-4.
152. Strukchinsky V. Express-request-id (GitHub Repository) [Electronic resource].
153. Tiskin A. The bulk-synchronous parallel random access machine // Theor. Comput. Sci. 1998. Vol. 196, no. 1–2. P. 109–130. DOI:10.1016/S0304-3975(97)00197-7.
154. Tiskin A. BSP (Bulk Synchronous Parallelism) // Encyclopedia of Parallel Computing. Boston, MA: Springer US, 2011. P. 192–199. DOI:10.1007/978-0-387-09766-4_311.
155. Touyama T., Horiguchi S. Performance evaluation of practical parallel computation model LogPQ // Proceedings of the Fourth International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'99). Washington, DC, USA: IEEE Computer Society, 1999. P. 216–221. DOI:10.1109/ISPAN.1999.778942.
156. Touyama T., Horiguchi S. Parallel computation model LogPQ // High Performance Computing. ISHPC 1997. Lecture Notes in Computer Science, vol 1336 / ed. Polychronopoulos C., Joe K., Araki K. A.M. Berlin, Heidelberg: Springer, 1997. P. 327–334. DOI:10.1007/BFb0024227.
157. Tu B. et al. Accurate Analytical Models for Message Passing on Multi-core Clusters // 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing. IEEE, 2009. P. 133–139. DOI:10.1109/PDP.2009.18.
158. Tu B. et al. Performance analysis and optimization of MPI collective operations on multi-core clusters // J. Supercomput. Springer US, 2012.

Vol. 60, no. 1. P. 141–162. DOI:10.1007/s11227-009-0296-3.
159. Unnebäck L. Multer (GitHub Repository) [Electronic resource].
160. Valiant L.G. General Purpose Parallel Architectures // Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity. Elsevier, 1990. P. 943–971. DOI:10.1016/B978-0-444-88071-0.50023-0.
161. Valiant L.G. A bridging model for parallel computation // Commun. ACM. 1990. Vol. 33, no. 8. P. 103–111. DOI:10.1145/79173.79181.
162. Valiant L.G. A bridging model for multi-core computing // J. Comput. Syst. Sci. Elsevier Inc., 2011. Vol. 77, no. 1. P. 154–166. DOI:10.1016/j.jcss.2010.06.012.
163. Vitter J.S., Shriver E.A.M. Algorithms for parallel memory, II: Hierarchical multilevel memories // Algorithmica. Springer-Verlag, 1994. Vol. 12, no. 2–3. P. 148–169. DOI:10.1007/BF01185208.
164. Wu K. et al. Early Evaluation of Intel Optane Non-Volatile Memory with HPC I/O Workloads // arXiv:1708.02199v2 [cs.DC]. 2017. 6 p.
165. Yang C.-T., Huang C.-L., Lin C.-F. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters // Comput. Phys. Commun. North-Holland, 2011. Vol. 182, no. 1. P. 266–269. DOI:10.1016/J.CPC.2010.06.035.
166. Yang K., Murty K.G. New iterative methods for linear inequalities // J. Optim. Theory Appl. Kluwer Academic Publishers-Plenum Publishers, 1992. Vol. 72, no. 1. P. 163–185. DOI:10.1007/BF00939954.
167. Yang X.I.A., Mittal R. Acceleration of the Jacobi iterative method by factors exceeding 100 using scheduled relaxation // J. Comput. Phys. Academic Press, 2014. Vol. 274. P. 695–708. DOI:10.1016/J.JCP.2014.06.010.
168. Yuan L. et al. LogGPH: A Parallel Computational Model with Hierarchical Communication Awareness // Proceedings of the 2010 13th IEEE International Conference on Computational Science and Engineering - CSE'10. Washington, DC, US: IEEE Computer Society, 2010. P. 268–274. DOI:10.1109/CSE.2010.40.
169. Zaslavski A.J. Approximate Solutions of Common Fixed-Point Problems. Cham: Springer International Publishing, 2016. Vol. 112. DOI:10.1007/978-3-319-33255-0.
170. Zhang Y. et al. Models of parallel computation: a survey and classification // Front. Comput. Sci. China. Higher Education Press, 2007. Vol. 1, no. 2. P. 156–165. DOI:10.1007/s11704-007-0016-1.
171. Бурцев А.П. Параллельная обработка данных сейсморазведки с использованием расширенной модели Master-Slave // Суперкомпьютерные дни в России: Труды международной конференции (26-27 сентября 2016 г., г. Москва). Москва: Изд-во МГУ, 2016. P. 887–895.
172. Воеводин В.В., Воеводин В.В. Параллельные вычисления. Санкт-

Петербург: БХВ-Петербург, 2002. 608 p.
173. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. 2nd ed. Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского, 2003. 184 p.
174. Ежова Н.А., Соколинский Л.Б. Модель параллельных вычислений для многопроцессорных систем с распределенной памятью // Вестник ЮУрГУ. Серия Вычислительная математика и информатика. 2018. Vol. 7, no. 2. P. 32–49. DOI:10.14529/cmse180203.
175. Ежова Н.А., Соколинский Л.Б. Исследование масштабируемости итерационных алгоритмов при суперкомпьютерном моделировании физических процессов // Вычислительные методы и программирование. 2018. Vol. 19, no. 4. P. 416–430. DOI:10.26089/NumMet.v19r437.
176. Ежова Н.А., Соколинский Л.Б. Обзор моделей параллельных вычислений // Вестник ЮУрГУ. Серия Вычислительная математика и информатика. 2019. Vol. 8, no. 3. P. 58–91. DOI:10.14529/cmse190304.
177. Ежова Н.А., Соколинский Л.Б. Модель параллельных вычислений BSF-MR // Системы управления и информационные технологии. 2019. no. 3(77). P. 15–21.
178. Ежова Н.А., Соколинский Л.Б. Верификация модели параллельных вычислений BSF-MR на примере гравитационной задачи // Параллельные вычислительные технологии (ПаВТ’2019). Короткие статьи и описания плакатов XIII Международной научной конференции. ISBN 978-5-696-05020-1. Челябинск: Издательский центр ЮУрГУ, 2019. P. 239–250.
179. Ежова Н.А., Соколинский Л.Б. Программная поддержка модели BSF // Вестник ЮУрГУ. Серия Вычислительная математика и информатика. 2019. Vol. 8, no. 4.
180. Ким А.К., Перекатов В.И., Фельдман В.М. На пути к российской экзасистеме: планы разработчиков аппаратно-программной платформы “Эльбрус” по созданию суперкомпьютера экзафлопсной производительности // Вопросы радиоэлектроники. 2018. no. 2. P. 6–13.
181. Коньшин И.Н. Модели параллельных вычислений для оценки реального ускорения исследуемого алгоритма // Суперкомпьютерные дни в России: Труды международной конференции (26-27 сентября 2016 г., г. Москва). Москва: Изд-во МГУ, 2016. P. 269–280.
182. Лупин С.А., Посыпкин М.А. Технологии параллельного программирования. Москва: ИД «ФОРУМ»: ИНФРА-М, 2011. 208 p.
183. Одинцов И.О., Тютляева Е.О., Московский А.А. На пути к инфраструктуре экзаскейльного суперкомпьютера // Научный сервис в сети Интернет: труды XIX Всероссийской научной конференции (18-23 сентября 2017 г., г. Новороссийск). Москва: ИПМ им.

М.В.Келдыша, 2017. P. 377–378. DOI:10.20948/abrau-2017-78.
184. Соколинская И.М., Соколинский Л.Б. О решении задачи линейного программирования в эпоху больших данных // Параллельные вычислительные технологии – XI международная конференция, ПаВТ'2017, г. Казань, 3–7 апреля 2017 г. Короткие статьи и описания плакатов. Челябинск: Издательский центр ЮУрГУ, 2017. P. 471–484.
185. Соколинская И.М., Соколинский Л.Б. Масштабируемый алгоритм для решения нестационарных задач линейного программирования // Вычислительные методы и программирование новые вычислительные технологии. 2018. Vol. 19, no. 4. P. 540–550. DOI:10.26089/NumMet.v19r448.
186. Сухорослов О.В. Развитие моделей программирования в GRID // Научный сервис в сети Интернет: решение больших задач. Труды Всероссийской научной конференции. Москва: Издательство Московского университета, 2008. P. 213–215.
187. EuroHPC - Leading the way in the European Supercomputing [Electronic resource]. URL: https://eurohpc-ju.europa.eu/ (accessed: 03.07.2019).
188. TOP500 Supercomputer Sites [Electronic resource]. URL: https://www.top500.org/ (accessed: 03.04.2017).
189. MPICH Documentation and Guides [Electronic resource].
190. Beowulf Cluster Computing with Linux / ed. Sterling T.L. Cambridge, London: MIT Press, 2002. 496 p.
191. Sourcebook of parallel computing / ed. Dongarra J. et al. San Francisco: Morgan Kaufmann, 2003. 842 p.
192. Executive Order -- Creating a National Strategic Computing Initiative [Electronic resource]. 2015. URL: https://obamawhitehouse.archives.gov/the-press-office/2015/07/29/executive-order-creating-national-strategic-computing-initiative (accessed: 02.07.2019).
193. Exascale Computing Project [Electronic resource]. 2016. URL: https://www.exascaleproject.org/ (accessed: 02.07.2019).
194. Кластерная архитектура РСК Торнадо [Electronic resource]. 2017. URL: http://www.rscgroup.ru/ru/our-technologies/360-klasternaya-arhitektura-rsk-tornado (accessed: 05.07.2019).
195. Docker Open Source Engine Guide. SUSE Linux Enterprise Server 15 SP1. SUSE LLC [Electronic resource]. 2019. P. 33. URL: https://documentation.suse.com/sles/15-SP1/pdf/book-sles-docker_color_en.pdf (accessed: 19.09.2019).