УДК 004.2+004.4 DOI: 10.17587/prin.13.3-16

В. В. Корнеев, д-р техн. наук, проф. гл. науч. сотр., korv@rdi-kvant.ru,
ФГУП "Научно-исследовательский институт "Квант", Москва

Параллельное программирование

Рассмотрен генезис моделей параллельного программирования. Показано, что параллелизм и аппаратная поддержка синхронизации, присущие архитектуре, обусловливают модель параллельного программирования. Современная элементная база требует перехода на модель программирования на базе разделяемой памяти.

Ключевые слова: *СБИС, архитектура, модель параллельного программирования, потоковая модель вычислений, граф межмашинных связей*

Введение

Несмотря на то что параллельные вычисления и модели параллельного программирования в настоящее время используются во всех сферах деятельности человека, принятие той или иной модели не осознается как технологический сдвиг, обусловливающий изменение архитектуры вычислительных систем, и, как следствие, модель параллельного программирования. Архитектура систем определяет возможности параллельного функционирования вычислительных ресурсов, а модель программирования — возможность эффективного отображения программы на аппаратные ресурсы.

В настоящей статье предпринята попытка рассмотреть генезис моделей параллельного программирования и обосновать необходимость перехода к модели с разделяемой памятью и синхронизацией на базе дополнительного бита слов памяти.

Однородные универсальные вычислительные системы высокой производительности

Параллельная обработка информации, как средство получения производительности, недостижимой для одиночного компьютера, возникла при решении актуальных вычислительно трудных задач на создаваемых для этого специализированных вычислительных системах. В начале 1960-х гг. в Институте математики имени С. Л. Соболева СО РАН (ИМ СОАН СССР) были начаты исследования по архитектуре вычислительных систем и вычислительных сред, параллельным алгоритмам и средствам параллельного программирования, машинной графике и распознаванию образов, что сейчас именуется искусственным интеллектом. Начальный этап исследований проблем построения и применения высокопроизводительных вычислительных систем завершился выходом в 1966 г. монографии [1]. Рассмотрим результаты, имеющие фундаментальное значение для появления и развития параллельного программирования, сведенные в монографии в единую концепцию на основе модели "коллектива вычислителей".

Впервые на основе анализа физических ограничений развития микроэлектроники были сформулированы подходы к повышению производительности вычислительных систем, включающие:

- увеличение тактовой частоты;
- рост числа одновременно (параллельно) функционирующих обрабатывающих устройств;
- программируемость структуры, как программную настройку связей между обрабатывающими устройствами.

Программируемость структуры при этом рассматривалась как средство достижения универсальности вычислительной установки, позволяющее получить высокую производительность при исполнении разных алгоритмов за счет программной реализации для исполнения этих алгоритмов специализированных вычислительных систем.

Чтобы вычислитель функционировал на тактовой частоте ν наибольшая длина связей (проводников) между устройствами не должна превышать $d_{\text{пр}} = c/2\xi\nu$, где c — скорость света; ν — тактовая частота единой сетки тактовых сигналов вычислителя; c/ν — длина волны, соответствующая тактовой частоте ν; ξ — коэффициент превышения длины волны над длиной связи (константа в пределах от 10 до 100). Пусть ρ — максимально допустимое число устройств в единице объема, определяемое технологией производства оборудования и возможностями теплоэнергообмена. Тогда число устройств, которое можно разместить в шаре диаметром $2d_{\text{пр}}$, не будет превышать $n_{\text{пр}} = 2\pi\rho c^3/3\xi^3\nu^3$. Производительность вычислителя прямо пропорциональна произведению числа устройств на тактовую частоту их функционирования. Поэтому предельная производительность вычислительных устройств с единой сеткой тактовых сигналов имеет принципиальное ограничение.

Следствие приведенной выше модели производительности заключается в том, что для достижения сколь угодно большой производительности нужно отказаться от единой тактовой сети и строить систему как совокупность синхронных вычислителей, объединенных асинхронными каналами. В этом случае ограничение на предельное число устройств действует только в пределах каждого отдельного вы-

числителя, а число таких вычислителей в системе не ограничено. Это обстоятельство позволяется достигать требуемой производительности, что и произошло в XXI веке. В настоящее время при построении систем их относят к архитектурам класса GALS (*Globally Asynchronous, Locally Synchronous*), в которых реализована глобальная асинхронность при передаче между синхронными вычислительными устройствами.

Доказанная возможность построения вычислительных систем с требуемым большим числом вычислительных машин, в терминологии авторов [1] — элементарных машин (ЭМ), еще не означала возможности решения задач с производительностью, пропорциональной числу использованных ЭМ. Для обоснования ограниченности роста производительности за счет параллельных вычислений использовались рассуждения, известные впоследствии как закон Амдаля: если в программе есть не распараллеливаемая часть σ и идеально распараллеливаемая часть $(1 - \sigma)$, то при исполнении этой программы на m процессорах достижимое за счет параллелизма ускорение $S = 1/(\sigma + (1 - \sigma)/m)$. В предельном случае при m, стремящемся к бесконечности, ускорение $S = 1/\sigma$.

Таким образом, с одной стороны, отсутствуют алгоритмизированные подходы к построению параллельных вычислительных систем и к разработке параллельных программ с доказанной эффективностью их исполнения. С другой стороны, есть прецеденты построения параллельных систем на базе параллельных алгоритмов решения вычислительно трудных задач. В этой ситуации было решено создать экспериментальную параллельную вычислительную систему "Минск-222" и начать исследовать на ее основе параллельные программы для актуальных вычислительных задач и наиболее часто встречающихся их фрагментов, реализующих вычислительные методы, например, решение систем линейных уравнений. В ходе этой работы предполагалось создать эффективные алгоритмы распараллеливания программ, в том числе — автоматизированного и автоматического.

Вычислительная система "Минск-222" была построена на базе серийной ЭВМ "Минск-22" путем подсоединения к каждой ЭВМ системного устройства (СУ), имеющего по два управляющих и информационных канала для соединения с соответствующими каналами двух соседних ЭВМ. В систему "Минск-222" могло быть объединено до 16 ЭМ, каждая имела номер i, $0 \leqslant i \leqslant N - 1$, N — число ЭМ в системе. Элементарная машина с номером i соединялась двунаправленными управляющим и информационным каналами с ЭМ с номерами k и j, $k = (i - 1) \bmod N$ и $j = (i + 1) \bmod N$. Таким образом, формировалась структура межмашинных связей с топологией "кольцо".

Каждое СУ имело в своем составе регистр настройки (РН), состоящий из триггеров: TR, TQ, TΩ, а также блок операций системы (БОС), расширяющий систему команд базовой ЭВМ так называемыми системными командами. Последние разделялись на четыре типа:

- команды настройки, управляющие установкой регистров настройки и регистров используемых признаков ЭМ;
- команды обобщенного безусловного перехода (ОБП), служащие для начальной загрузки программ и данных в ЭМ с установленным в "1" триггером TQ и принудительного управления ходом вычислений в этих ЭВМ (состояние "1" или "0" триггера TQ в ЭМ определяет соответственно выполнение или пропуск команды, выдаваемой из ЭМ, выполняющей команду ОБП);
- команды обобщенного условного перехода (ОУП), выполняющей переход по указанному в ней адресу, при условии, что

$$\Omega_k = \mathop{\&}_{i \in E} \omega_{ki}, \quad k = 1, 2, 3,$$

где E — подмножество номеров машин с предварительно установленными в "1" триггерами TΩ и выбранным признаком ω_k, $k \in \{1, 2, 3\}$, использованным для выработки обобщенного признака Ω_k, ω_{ki} — признак (результат меньше нуля, переполнение, результат равен нулю), вырабатываемый ЭМ с номером i;
- команды передачи (П) z слов памяти, начиная с заданного в команде адреса, и команды приема (ПР) w слов памяти с размещением их в собственной памяти, начиная с заданного в команде адреса, с выполнением их приема вплоть до поступления всех w слов и только после этого перехода к выполнению команды, следующей за командой приема.

Установка в "1" триггеров TR в ЭМ с номерами i и $i + n(\bmod N)$, а также установка в "0" триггеров TR в ЭМ с номерами $i + 1(\bmod N)$, ..., $i + n - 1(\bmod N)$, приводила к образованию подсистемы из n ЭМ с номерами i, $i + 1(\bmod N)$, ..., $i + n - 1(\bmod N)$. Таким образом, система могла быть разделена на совокупность подсистем, функционирующих как самостоятельные системы.

Архитектура системы "Минск-222" определила модель параллельного программирования на базе передачи сообщений и распределенной памяти. Усилиями специалистов по алгоритмам и программистов ИМ СОАН СССР были разработаны параллельные программы, настраиваемые на предоставленное число ЭМ, как на параметр. Такие программы удалось создавать для практически важных вычислительных задач и получать при их исполнении ускорение, прямо пропорциональное числу используемых ЭВМ с некоторым повышающим коэффициентом. Большая емкость оперативной памяти в системе "Минск-222" по сравнению с одной ЭВМ "Минск-22" с медленной внешней памятью на магнитной ленте и быстродействие каналов связи, сравнимое с быстродействием оперативной памяти, обеспечивали производительность, значительно превосходящую "механическую" сумму производительностей машин системы.

Сформировалось представление параллельной программы как совокупности ветвей с операторами взаимодействия между ними по данным и управлению. Фундаментальным результатом, полученным на основе опыта создания практически востребованных параллельных программ, было выявление типовых схем обмена данными между ветвями параллельной программы. Эти схемы сводятся к пяти типам обмена: трансляционный; трансляционно-циклический;

конвейерно-параллельный; коллекторный; дифференцированный. При трансляционном обмене один и тот же блок данных передается из одной (любой) ветви программы одновременно во все остальные ее ветви. Трансляционно-циклический обмен транслирует блок данных из каждой ветви во все остальные. Конвейерно-параллельный обмен обеспечивает передачу блока данных из каждой ветви i, выполняемой на ЭМ с номером i, в соседнюю ветвь k, $k = (i + 1) \bmod B$, где B — число ветвей параллельной программы. Коллекторный обмен реализует сбор блоков данных из других ветвей программы в одну ветвь. Наконец, дифференцированный обмен выполняет передачу блоков данных между задаваемыми парами ветвей или из одной ветви в несколько.

В середине 1990-х гг. модель программирования на базе передачи сообщений стала общепризнанной и оформилась как *Message Passing Interface* (MPI) — библиотека функций для создания и исполнения параллельных программ. При этом имеет место следующее соответствие между функциями системы параллельного программирования "Минск-222" и MPI:

- трансляционный обмен — MPI_Bcast;
- трансляционно-циклический обмен — MPI_Alltoall или MPI_Allscatter;
- коллекторный обмен — MPI_Gather;
- обобщенный условный переход — MPI_Barrier;
- дифференцированный обмен — MPI_Send, MPI_Recv.

Таким образом, можно констатировать, что в монографии [1] была предложена перспективная, со временем ставшая основополагающей, концепция архитектуры параллельных вычислительных систем с распределенной памятью и модель параллельного программирования на базе обмена сообщениями.

Начиная с 1970-х гг. предпринимались попытки создавать параллельные системы, базирующиеся на этой концепции. Так был проект объединения машин старших моделей ЕС ЭВМ, но в одной, даже крупной организации, была только одна ЭВМ, а объединение ЭВМ в пределах страны представляло собой не только техническую проблему. Кроме того, используемая элементная база позволяла создавать более производительные ЭВМ с единой сетью синхронизации с повышением тактовой частоты, так как число элементов в них было далеко до предела, определяемого плотностью упаковки элементов и тактовой частотой.

Положение изменилось с появлением микропроцессоров. Была создана вычислительная система МВС-1000М [2] с производительностью на уровне 10^{12} флопс. Это была первая система отечественной разработки, попавшая в верхние 50 позиций списка TOP500 самых производительных вычислительных установок в мире. Архитектура системы МВС-1000М предоставляет только системные операции передачи и приема блока данных. Как интерфейс прикладного программирования пользователям предоставляются вызовы программ библиотеки MPI. В определенной мере это является следствием отсутствия возможности добавить системные команды, как это сделано в "Минск-222", в микропроцессор. Однако программная реализация барьерной синхронизации в MPI, в отличие от системной команды ОУП "Минск-222", требует значительных временных затрат и служит одним из основных источников потери производительности при параллельных вычислениях. Поэтому, например, в IBM Blue Gene\L введена специальная коммуникационная сеть для реализации аналога ОУП "Минск-222".

Вычислительные системы с программируемой структурой

В начале 1970-х гг. появились теоретические вопросы по коренному противоречию между реализацией архитектуры "Минск-222" и моделью параллельных систем из монографии [1]. Вопросы заключались в том, что реализация предполагала "длинные" связи, например, при выполнении системной команды ОУП, что при увеличении числа ЭВМ системы должно было вести к снижению тактовой частоты. Кроме того, команды П и ПР предполагали наличие связей между любыми ЭМ системы, сколь большой бы она не была, т. е. связи должны быть "длинными". А архитектуры класса GALS, локально синхронные и глобально асинхронные, не должны иметь "длинных" связей. Каждая ЭМ должна иметь связи только с ограниченным числом соседних ЭМ, а для пересылки данных в не соседнюю ЭМ должна использоваться последовательность пересылок между соседними ЭМ. Поэтому требовалось понять, как строить вычислительные системы, имеющие только "короткие" локальные связи между ЭМ и как программировать и эффективно исполнять параллельные программы с производительностью, прямо пропорциональной числу используемых ЭМ. Результаты исследований по этим и смежным вопросам опубликованы в вышедшей в 1985 г. монографии [3]. Ниже будут представлены основные положения предложенной архитектуры и подхода к параллельному программированию с локальными связями.

Потенциальная неограниченность числа ЭМ, объединяемых в систему, может быть достигнута только при использовании пространственного распределенного коммутатора межмашинных связей с децентрализованным устройством управления. Во всяком другом случае структуру системы можно представить, как совокупность машин, связанных проводниками с одним коммутатором, что противоречит потенциальной неограниченности числа машин при неизменной тактовой частоте.

Пространственный коммутатор $N \times N$ с децентрализованным управлением с N входами и N выходами строится из N неординарных коммутаторов с v + 1 входом $\mathbf{вх}_i$ и выходом $\mathbf{вых}_i$, $i = 0, 1, ..., v$. Пример построения такого коммутатора с восемью входами и выходами, созданного на базе коммутаторов с v = 3, показан на рис. 1. При этом вход $\mathbf{вх}_0$ и выход $\mathbf{вых}_0$ каждого из N неординарных коммутаторов j, $j = 1, ..., N$, служат $Вх_j$ и $Вых_j$ пространственного коммутатора.

При создании вычислительной системы из N машин и имеющихся коммутаторов с v + 1 входом $\mathbf{вх}_i$

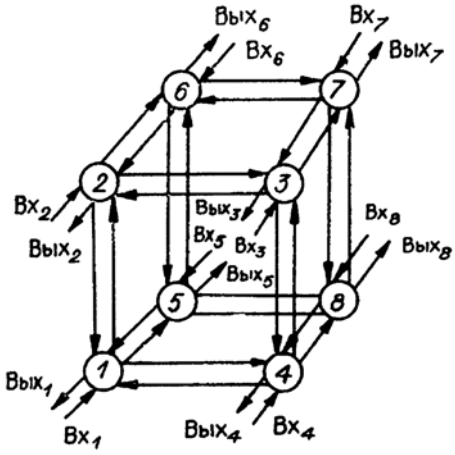

Рис. 1. Пространственный коммутатор 8×8

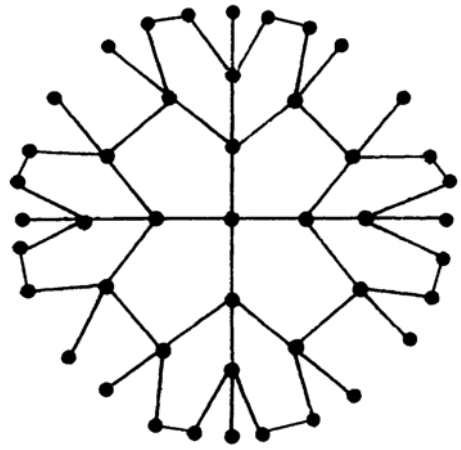

Рис. 2. Первые три яруса бесконечного планарного графа *L*(4, 5)

и выходом **вых**$_i$, i = 0, 1, ..., v, необходимо выбрать граф межмашинных связей. Поскольку существуют различные неизоморфные однородные графы с числом вершин N и степенью вершин v, возникает проблема выбора из них графа, оптимального по критериям, предъявляемым к графам межмашинных связей. Была выдвинута гипотеза, подтвержденная статистическими экспериментами, что оптимальный граф обладает минимальными диаметром и средним диаметром среди графов с числом вершин N и степенью вершин v.

В качестве графов межмашинных связей предложено параметрическое семейство однородных графов Л(N, v, g) с количеством вершин N, степенями вершин v, обхватом g, каждая вершина которых входит в v циклов длины g.

Граф Л(N, v, g) строится на основе N-вершинного подграфа бесконечного планарного графа L(v, g) путем выбора соответствующего подграфа из всех существующих и формирования ребер с образованием графа Л(N, v, g). При этом N-вершинный подграф включает:

- все вершины d ярусов графа L(v, g), если $\sum_{i=0}^{d} Ni = N$, где Ni — число вершин на ярусе i, i = 0, ..., d, N_0 = 1, N_1 = v, d — диаметр строящегося графа Л(N, v, g);
- все вершины d − 1 ярусов графа L(v, g), если $\sum_{i=0}^{d-1} Ni < N$ и $N - \sum_{i=0}^{d-1} Ni$ вершин яруса d.

Алгоритм перебирает возможные варианты и либо находит требуемый граф, либо определяет отсутствие возможности его построения. На рис. 2 приведены три яруса бесконечного планарного графа L(4, 5).

Графы Л(N, v, g) существуют не для всех комбинаций значений параметров N, v и g. С помощью переборного алгоритма удалось, например, построить графы Л(N, 4, 5) для значений N = 19, ..., 45.

На рис. 3 приведены графы Л(10, 3, 5) и Л(24, 4, 5).

Граф Л(10, 3, 5) имеет диаметр 2 и включает все вершины ярусов 0, 1, 2 графа L(3, 5), Граф Л(24, 4, 5) имеет диаметр 3 и включает все вершины ярусов 0, 1, 2 графа L(4, 5), а также 7 вершин яруса 3 этого графа.

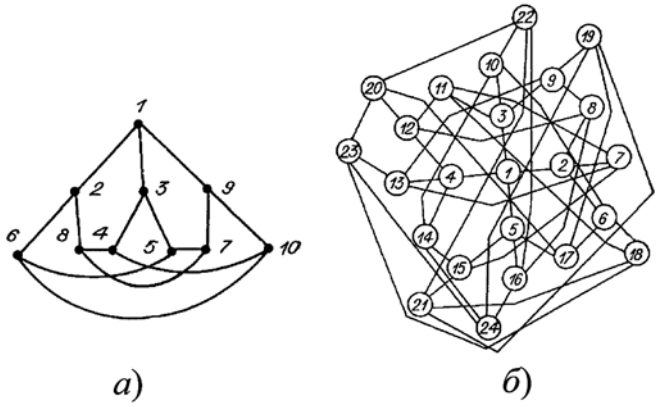

Рис. 3. Графы Л(10, 3, 5) (*а*) и Л(24, 4, 5) (*б*)

Графы Л(N, 4, 4) являются групп-графами конечных абелевых групп порядка N с двумя образующими элементами. Они существуют для всех значений N.

У графа Л(N, v, g) будут минимальные диаметр d и средний диаметр $d_{ср}$, если $g = 2d + 1$ или $g = 2d$ среди всех графов с числом вершин N, степенями вершин v.

Возможной постановкой задачи поиска оптимального графа межмашинных связей служит ограничение диаметра (числа хопов — промежуточных передач пакетов). В этом случае для построения используются коммутаторы с v ⩾ 64. В работе [4] представлены алгоритмы синтеза графов, близких к Л(N, v, 5) и Л(N, v, 7), с диаметрами 2 и 3, соответственно.

В работе [5] представлены результаты статистического моделирования, доказывающие оптимальность использования Л(N, v, g) с минимальными диаметром d и средним диаметром $d_{ср}$, включая эффективность реализации функций библиотеки MPI на вычислительных системах с графами межмашинных связей Dragonfly и Л(N, v, 7) с одинаковыми числом вершин и степенями вершин. С помощью собственного переборного алгоритма синтезированы графы Л(20, 4, 7), Л(30, 5, 7), Л(36, 5, 7), Л(252, 11, 7), Л(264, 11, 7) с диаметром 3.

Важными прикладными аспектами использования Л(N, v, g) с минимальными диаметром d и средним диаметром $d_{ср}$ в качестве графа межмашинных связей служат:

- максимизация вероятности образования связного подграфа из исправных машин при заданных вероятностях отказа машин и линий связи среди всех графов с числом вершин N и степенями вершин v (структурная живучесть);
- максимизация вероятности числа одновременно устанавливаемых соединений между отдельными машинами и/или группами машин (структурная коммутируемость).

Архитектура с локальными связями представляет возможность непосредственной передачи блока данных только между соседними машинами, соединенными линией связи. Структура ЭМ вычислительной системы с графом Л(N, 4, 4) показана на рис. 4.

В передающей машине блок данных помещается в выходную очередь линии связи. Если во входной очереди этой линии связи имеется место для приема блока данных, то он поступает во входную очередь. Если во входной очереди нет свободного места, то передача задерживается вплоть до момента освобождения места во входной очереди.

Передача блока данных между машинами, не соединенными линией связи, реализация ОУП, а также ОБП выполняется через машины, служащие соседними друг другу. Выделение подсистем (программирование структуры системы) выполняется программно средствами системного программного обеспечивания. С точки зрения программиста, структура, для которой создается параллельная программа, выбирается из числа представленных на рис. 5 или какая-либо другая. Важно чтобы эта структура имела параметрическое описание и могла быть выделена с любым требуемым числом машин. Следует отметить, что "корневое дерево" может быть создано на связном подграфе при любом Л(N, v, g) графе межмашинных связей. Структуры "линейка" и "кольцо" при этом имеют ограничения на возможности их построения, а "решетка" строится без использования транзитных машин только на Л(N, v, 4) графе.

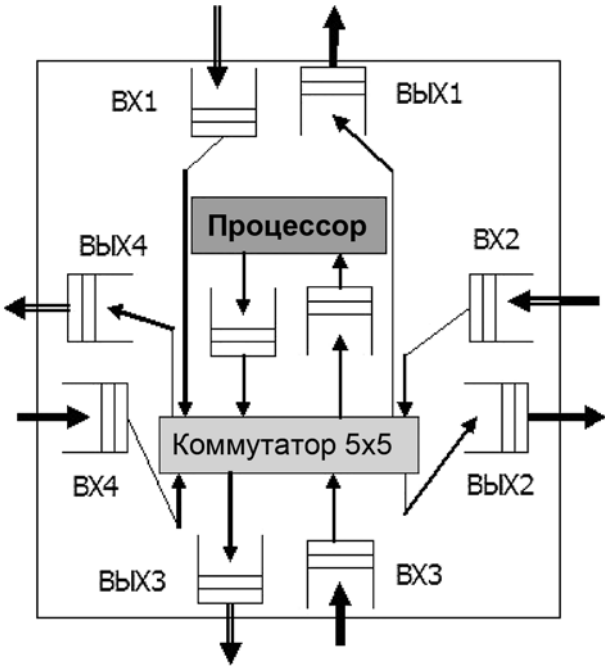

Рис. 4. Структура ЭМ вычислительной системы с графом Л(N, 4, 4)

Рис. 5. Структуры подсистем:
а — "линейка"; *б* — "кольцо"; *в* — "корневое дерево"; *г* — "решетка"

Для возможности программирования необходимо определить нумерацию машин. Будем полагать, что при построении подсистемы выбирается корневая машина, которой присваивается номер 0. Нумерация машин для подсистем "линейка" и "кольцо" естественна, для подсистемы "решетка" могут быть варианты по строкам или столбцам. Примем, что нумерация машин подсистемы "корневое дерево" выполняется по алгоритму "поиск в глубину" с учетом одного и того же порядка номеров полюсов для всех коммутаторов, образующих пространственный коммутатор системы. Примеры задания такой нумерации приведены на рис. 6.

Параллельная программа должна задавать поток данных между локально взаимодействующими машинами, которые исполняют локально свои параллельные ветви. Каждый обмен данными между ветвями параллельной программы выполняется по одной из ранее рассмотренных схем: трансляционной, трансляционно-циклической, конвейерно-параллельной, коллекторной или дифференцированной. По этой причине параллельная программа должна состоять из двух взаимодействующих программ — вычислительной и путевой процедур.

Рассмотрим суть этих процедур. Пусть требуется выполнить трансляционно-циклическую схему обмена. На рис. 7 показана подсистема из шести машин со структурой "корневое дерево", а также сформированные для реализации этой схемы обмена "восходящая" и "нисходящая" сети подсистемы. Будем обозначать τ_i вес вершины i, равный числу вершин поддерева, корнем которого она служит, для подсистемы рис. 7, $\tau_0 = 6$, $\tau_1 = 1$, $\tau_2 = 4$.

Путевая процедура, реализующая трансляционно-циклический обмен, задает функционирование восходящей и нисходящей сетей подсистемы, на $n + 1$ машинах которой исполняется параллельная программа.

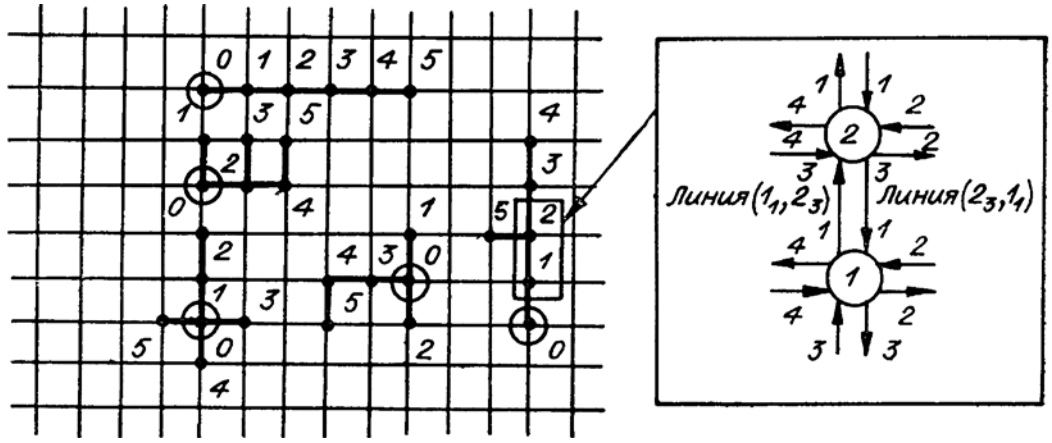

Рис. 6. Подсистемы из шести машин с заданными номерами (обведены корни деревьев)

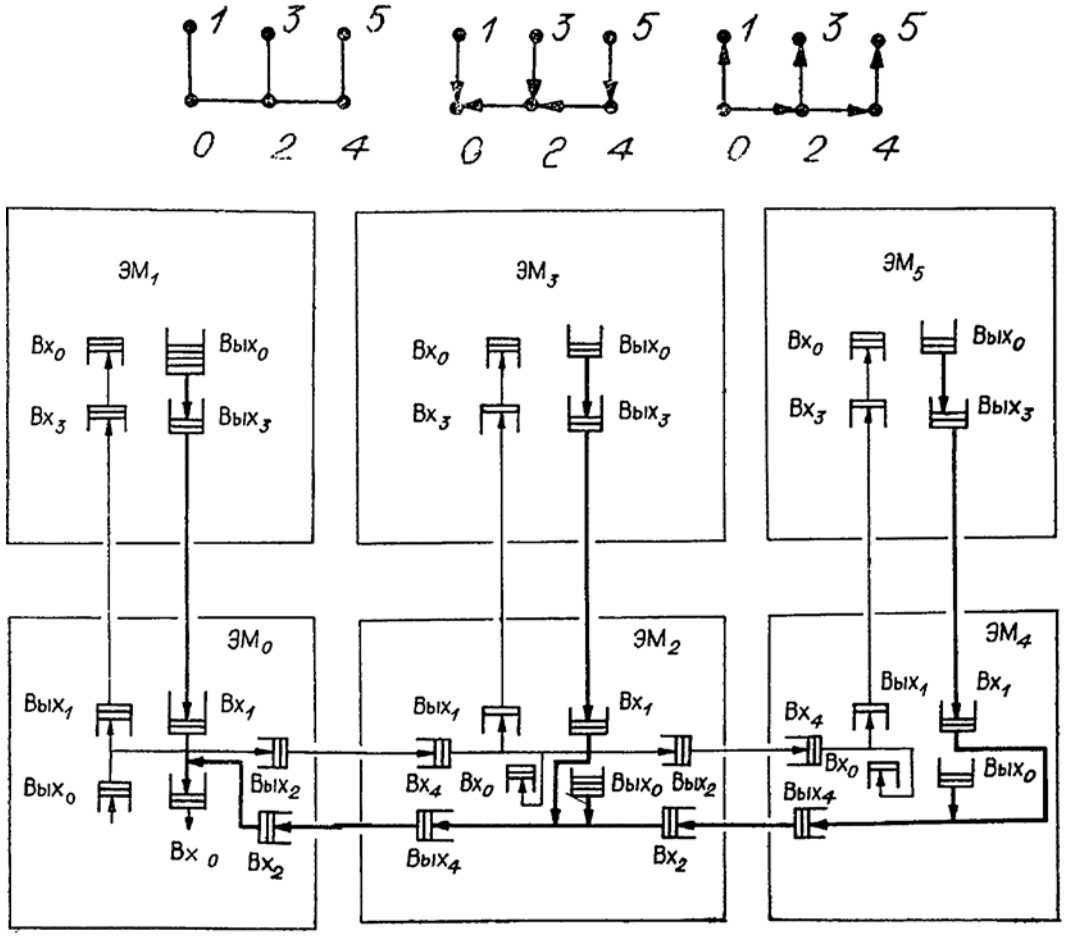

Рис. 7. Подсистема из шести машин со структурой "корневое дерево"

В восходящей сети каждая машина $ЭМ_i$, $i = 1, ..., n$, загружает по следующей циклической дисциплине очередь $Вых_{e(j)}$ линии (i, j), передающую элемент данных в $ЭМ_j$, служащую для нее машиной-преемником в восходящей сети, $e(j)$ — номер полюса коммутатора, $e(j) \in \{1, ..., v\}$ линии (i, j). Первым передается один элемент данных, выбираемый из начала очереди $Вых_0$, вырабатываемый в самой $ЭМ_i$, $i = 1, ..., n$. Следом за ним в $Вых_{e(j)}$ заносятся τ_{i1} элементов данных, каждый из которых выбирается из начала очереди $Вх_{i1}$, содержащей элементы данных, поступающие в $ЭМ_i$ из $ЭМ_{i1}$. $ЭМ_{i1}$ служит машиной-предшественником $ЭМ_i$ и имеет наименьший номер $i1$ среди всех машин $ЭМ_{i1}$, ..., $ЭМ_{ip}$, являющихся

машинами-предшественниками $ЭМ_i$ в восходящей сети $i_1 < ... < i_p$. После τ_{i1} элементов данных из очереди $Вх_{i1}$ в $Вых_{e(j)}$ заносятся τ_{i2} элементов данных из очереди $Вх_{i2}$ и далее, пока не будет занесено τ_{ip} из очереди $Вх_{ip}$. Затем последовательность действий повторяется, начиная с передачи одного элемента данных, выбираемого из начала очереди $Вых_0$.

Корневая машина $ЭМ_0$ загружает очередь $Вх_0$, циклически выбирая по описанной выше схеме элементы данных из очередей $Вх_{01}$, $Вх_{02}$, ..., $Вх_{0s}$, поступающие в $ЭМ_0$ из машин-предшественников $ЭМ_{01}$, ..., $ЭМ_{0s}$.

Функционирование нисходящей сети инициируется $ЭМ_0$. Из начала очереди $Вых_0$ выбирается элемент данных, выработанный вычислительной процедурой в $ЭМ_0$, и загружается в конец очередей $Вых_{01}$, $Вых_{02}$, ..., $Вых_{0s}$ для передачи по нисходящей сети из $ЭМ_0$ в машины-преемники $ЭМ_{01}$, ..., $ЭМ_{0s}$. Каждая $ЭМ_i$ в нисходящей сети заносит выбранный из начала очереди $Вх_{e(j)}$ элемент данных, поступивший из $ЭМ_j$ машины-предшественника, в свою очередь $Вх_0$ и выходные очереди, пересылающие элемент данных в машины-преемники.

Вычислительная процедура, исполняемая каждой машиной, берет элементы данных из очереди $Вх_0$ и после обработки помещает результат в очередь $Вых_0$. Если какая-либо ЭМ не завершила вычислительную процедуру к моменту, когда она может загружать элемент данных, то передача задерживается, равно как если в какой-либо очереди нет места для приема элемента данных.

Совместное функционирование восходящей и нисходящей сетей приводит к загрузке в очередь $Вх_0$ в каждой $n + 1$ машине подсистемы последовательности $x_1^0, x_2^0, ..., x_n^0, x_1^1, ..., x_n^1, x_1^2, ...$.

Используя приведенную выше реализацию трансляционно-циклического обмена, можно, слегка ее модифицировав и создав соответствующую вычислительную процедуру, строить параллельные программы, применяющие другие схемы обменов. Реализация дифференцированных обменов возможна, например, с использованием подобно трансляционно-циклической схемы модифицированной восходящей и нисходящей сетей. Модификация восходящей сети состоит в том, что циклически опрашиваются все очереди, поставляющие элементы данных, пропуская пустые. Сами элементы данных снабжаются тэгом, состоящим из номера ЭМ, их выработавшей, и номеров ЭМ, назначенных получателями.

Другая реализация межмашинных обменов основана на адресации машин подсистемы, исполняющей ветви параллельной программы.

Для реализации дифференцированных обменов может использоваться задание адресов машин и передача элементов данных по кратчайшим путям адресатам. Известна координатная адресация, которая применяется в вычислительных системах с Л(N, v, 4) графом или многомерными кубическими структурами в качестве графов межмашинных связей, в которых адрес ЭМ задается координатами по каждому направлению. При передаче элемента данных сравниваются значения координат адреса назначения и ЭМ, в которой выполняется сравнение. Если значения всех координат равны, то адресат достигнут. Среди направлений передачи с не совпавшими координатами выбирается ведущее к уменьшению рассогласования. Это эффективная адресация. Она используется в ряде суперкомпьютеров, например, фирмы CRAY.

Для вычислительных систем с Л(N, v, g) — графом межмашинных связей предложена Д(z, m)-адресация, при которой адрес $ЭМ_i$, $i \in \{0, ..., n-1\}$, также задается вектором $\mathbf{A}_i = A_{i0}, ..., A_{im-1}$. Каждый A_{ij}, $j = 0, ..., n-1$, принимает значение из множества $\{0, 1, ..., 2^z - 1\}$, $z \in \{1, 2, ...\}$, n — количество машин подсистемы, $mz \geqslant -\lceil \log_2 n \rceil$, $\lceil x \rceil$ — наименьшее целое такое, что $x \geqslant \lceil x \rceil$. $ЭМ_i$ и $ЭМ_k$, $i, k \in \{0, 1, ..., n-1\}$, принадлежат подсистеме $R_r(A_{i0}, ..., A_{ir-1})$ яруса r, если $A_{ir} \neq A_{kr}$, и $A_{ij} = A_{kj}$ для всех $j < r$, $r = 1, 2, ..., m$; $R_0()$ — подсистема яруса 0 (вся подсистема из n машин), $R_m(A_{i0}, ..., A_{im-1})$ — подсистема яруса m, состоящая из одной $ЭМ_i$. Адресация машин должна удовлетворять следующим свойствам:

- машины каждой подсистемы $R_r(A_{i0}, ..., A_{ir-1})$ яруса r, $r = 1, 2, ..., m$, должны вместе с линиями связи между ними отображаться в связный подграф графа межмашинных связей;
- $R_0 \supseteq R_1(A_{i0}) \supseteq R_2(A_{i0}, A_{i1}) \supseteq ... \supseteq R_m(A_{i0}, ..., A_{im-1})$;
- $R_r(A_{i0}, ..., A_{ir-1}) = \bigcup_{j=0}^{d} R_{r+1}(A_{i0}, ..., A_{ir-1}, j)$, $d = 2^z - 1$;
- $R_{r+1}(A_{i0}, ..., A_{ir-1}, j) \cap R_{r+1}(A_{i0}, ..., A_{ir-1}, k) = \varnothing$, $j \neq k$.

Подсистема из 15 машин с заданной Д(2, 3)-адресацией показана на рис. 8.

При Д(z, m)-адресации в каждой машине $ЭМ_i$, $i \in \{0, 1, ..., n-1\}$, должно храниться только $m2^z$ номеров $p_1(0)$, $p_1(1)$, ..., $p_1(2^z - 1)$, $p_2(0)$, $p_2(1)$, ..., $p_2(2^z - 1)$, ..., $p_m(0)$, $p_m(1)$, ..., $p_m(2^z - 1)$ выходных полюсов $ЭМ_i$, принадлежащих кратчайшим путям из $ЭМ_i$ соответственно в подсистемы $R_1(0)$, $R_1(1)$, ..., $R_1(2^z - 1)$, $R_2(A_{i0}, 0)$, ..., $R_2(A_{i0}, 2^z - 1)$, ..., $R_m(A_{i0}, ..., A_{im-2}, 2^z - 1)$. Кратчайший путь до подсистемы — это путь до ближайшей машины этой подсистемы. Минимальный

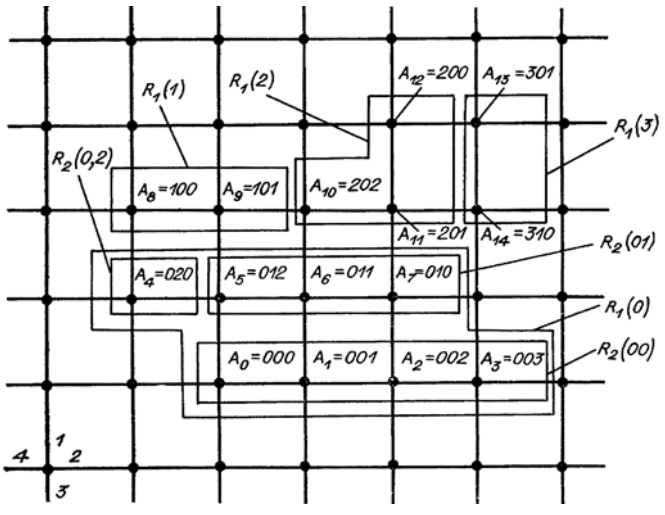

Рис. 8. Подсистема из 15 машин с Д(2, 3)-адресацией

объем требуемой памяти достигается при z = 1 и $m = \lceil \log_2 n \rceil$.

Одним из вариантов реализации Д(z, m) — адресации служит следующий. Полагаем m равным расстоянию от корневой до наиболее удаленной от нее машины подсистемы. Корневая машина $ЭМ_0$ получает адрес $A_0 = A_{00}$, ..., $A_{0m-1} = 0$, ..., 0. Остальные машины получают адреса, состоящие из последовательности номеров выходных полюсов ЭМ, принадлежащих пути между корневой и адресуемой машинами. Если длина этого пути r, $r = 1, 2, ..., m$, меньше m, то компоненты адреса, начиная с r-й, равны 0.

При Д(z, m)-адресации определение достижения адресата и выбор направления дальнейшей передачи выполняются по тому же алгоритму как при координатной адресации, за исключением того, что при выборе направления передачи учитывается рассогласование координаты с наименьшим номером. Передачи выполняются по близким к кратчайшим путям, так как хранятся направления к подсистемам, а не к требуемому адресату.

На базе Д(z, m)-адресации могут быть построены путевые процедуры для эффективной реализации трансляционной, трансляционно-циклической, конвейерно-параллельной, коллекторной схем обмена.

С использованием рассмотренных путевых и вычислительных процедур для широкого круга вычислительных методов были созданы параллельные программы с ускорением вычислений, пропорциональным числу используемых машин вычислительной системы с архитектурой с локальными связями. Этот результат достигается за счет программирования структуры, необходимой для реализации схем обмена исполняемого параллельного алгоритма. Поэтому вычислительные системы с архитектурой с локальными связями логично называть "вычислительные системы с программируемой структурой".

Для проверки всего комплекса идей по построению программно-аппаратных средств объединения машин в систему, децентрализованной распределенной операционной системы, средств программирования структуры и разработки параллельных программ был реализован макет вычислительной системы с программируемой структурой МИКРОС. Система построена на базе серийных микроЭВМ "Электроника-60". Общий вид системы показан на рис. 9.

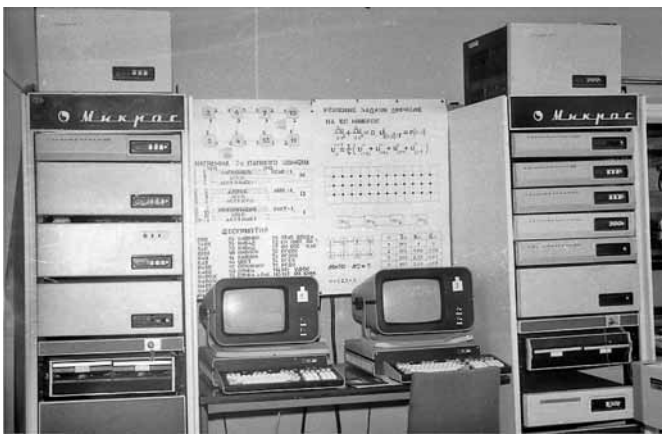

Рис. 9. Вычислительная система с программируемой структурой МИКРОС, 1986 г.

Экспериментальные исследования, проведенные на системе МИКРОС, продемонстрировали реализацию основных положений концепции вычислительных систем с программируемой структурой, что было подтверждено актом комиссии СО АН СССР, утвержденным председателем Президиума СО АН СССР.

К настоящему времени параллельные вычисления с локальными связями между машинами реализуются в основном в спецвычислителях с фиксированной структурой и известны как систолические и волновые. Программирование современных суперкомпьютеров, построенных на СБИС с небольшим (в пределах сотен) числом процессорных элементов, выполняется, как правило, с использованием MPI. Однако представляется, что при использовании СБИС с 10^3 и более процессорных элементов программируемость структуры станет практически востребованной как средство достижения универсальности параллельного программирования. Возможно, что это будет внутри MPI, а прикладной программист будет пользоваться привычным интерфейсом.

Потоковые вычислительные системы с разделяемой памятью

Исходя из современных представлений о развитии элементной базы и архитектур, суперкомпьютер экзафлопсного уровня производительности должен обрабатывать порядка 10^9 и более потоков при производительности одного потока порядка 10^9флопс. Однако практика использования современных компьютеров списка TOP500 показывает, что уже на компьютерах с миллионами потоков проявляются препятствия их эффективному применению и дальнейшему повышению быстродействия в силу простоев процессоров в ожидании завершения чтения/записи данных (так называемая "стена памяти").

В концептуальном аспекте, преодоление разрыва между временем выполнения команд в процессоре и временем доступа к памяти возможно при создании программ, использующих весь присущий алгоритму параллелизм до мельчайших гранул. В этом случае появляется возможность при ожидании одним потоком завершения обращения к памяти выполнять другие потоки, параллельные с ним [6], т. е. работать с памятью в темпе обслуживания потока запросов. Для снижения потерь производительности при переключении процессора с исполнения одного потока на выбор и начало исполнения другого предложены "легкие" потоки — q-треды, имеющие минимальный контекст. При таком подходе программа для суперкомпьютера экзафлопсного уровня производительности должна представлять собой описание порождения миллиардов потоков, межпотоковых коммуникаций и синхронизации этих потоков. Эту программу компилятор (статически) и библиотеки (динамически в ходе вычислений) преобразуют в совокупность потоков, определяя ресурсы, на которых будут протекать эти потоки или оставляя отображение потоков на ресурсы Runtime системы времени исполнения.

Могут использоваться отображения как в асинхронные потоки, протекающие в универсальных процессорах и имеющие собственный отдельный счетчик команд, так и в синхронные потоки, выполняемые по одному счетчику команд в разных арифметическо-логических устройствах (АЛУ), например, в GPGPU NVidia Fermi синхронно в каждом такте может протекать до 512 потоков. Синхронные потоки составляют аппаратно экономную альтернативу асинхронным, но более сложны в программировании [7]. Таким образом, программа будет исполняться на гетерогенных ресурсах. Необходимо для доступных заданию гетерогенных ресурсов обеспечить оптимальное назначение на них тредов задания. Конечной целью служит автоматическое назначение, но начинать следует с создания API библиотеки формирования требуемой совокупности ресурсов и назначения тредов на эти ресурсы. Надо уметь определять гетерогенные ресурсы, например, устройства выполнения пучков синхронных тредов, и выделять в программе треды, которые должны выполняться в синхронном режиме на этих устройствах. В ходе отображения тредов на ресурсы должны формироваться межтредовые коммуникации с использованием наиболее подходящего коммуникационного ресурса и режимов использования этих ресурсов (организация потоков сообщений, формирование сообщений заданной длины путем сборки из нескольких и т. д.).

Существующая модель программирования на базе передачи сообщений не позволяет эффективно использовать потенциальный параллелизм алгоритмов обработки. Создание других моделей экзафлопсного программирования необходимо чтобы:

— эффективно использовать многократно возросший параллелизм обработки (10^9 и более потоков);

— при обеспечении высокой степени параллелизма не вносить дополнительных трудностей при разработке параллельных программ, т. е. не снижать продуктивности их разработки.

Более того, новые модели должны многократно в сравнении с существующим уровнем повышать эту продуктивность. Этому должно способствовать применение глобально адресуемой памяти, а также повышение уровня и непроцедурности средств параллельного программирования, которые характерны для новых моделей.

В качестве основы модели экзафлопсного программирования, удовлетворяющей вышеуказанным требованиям, может быть выбрана модель, известная как *Parallel Random Access Model* (PRAM) [8]. Эта модель базируется на архитектуре мультипроцессора с общей разделяемой памятью. Процессоры P_i, $i = 1, ..., n$, имеют доступ к общей памяти (*distributed shared memory* — DSM) или секционированной общей памяти (*partitioned global address space* — PGAS), которая физически распределена по вычислительным модулям (процессор + блок памяти), однако имеет общее адресное пространство и логически доступна всем процессорам. Создавая программу, пользователь предполагает, что команды потоков, протекающие в каждом из процессоров, выполняются синхронно (время выполнения всех команд одно и то же), а межпроцессовые коммуникации и синхронизация осуществляются через ячейки разделяемой памяти.

Возможны различные подходы к разрешению конфликтов доступа разных процессоров к одной ячейке разделяемой памяти [8]. В контексте настоящего рассмотрения будем полагать приемлемым подход *concurrent-read exclusive-write* (CREW) с возможностью чтения одной и той же ячейки совокупностью процессоров, а записи только одним из процессоров. При этом все одновременные доступы к памяти по чтению предшествуют доступу по записи.

В этой модели параллельного программирования пользователь должен указывать, какие вычисления можно проводить параллельно, сообразуясь только с выбранным алгоритмом. Это способствует выявлению всего параллелизма, присущего алгоритму, и повышает продуктивность программирования [8].

Механизм разрешения конфликта доступа к одной ячейке разделяемой памяти CREW реализуется аппаратными средствами управления доступом к памяти. В свое время именно отсутствие эффективного масштабируемого аппаратного решения для разрешения конфликтов доступа к памяти привело к отказу от PRAM и переходу к модели на базе передачи сообщений. Однако представляется, что такое решение в настоящее время существует.

Рассмотрим по порядку отдельно языковые средства порождения и завершения процессов и средства межпроцессовых коммуникаций и синхронизации в рамках предлагаемого расширения PRAM-модели [8].

Языком параллельного программирования, реализующим модель PRAM, может служить расширение языка C [8]. Для порождения и завершения асинхронных тредов в него введены три дополнительные функции: spawn (*a*, *n*), join, fetch_and_add(*e*, *x*). Эти функции позволяют, соответственно:

- породить заданное параметром *n* число тредов с последовательными номерами, начиная с номера *a*;
- завершить тред, исполняющий join, и перейти к следующему оператору, если завершены все порожденные соответствующей spawn треды;
- выполнить как неделимую атомарную операцию присвоения переменной, например, номеру треда, значения параметра *x* и увеличить значение *x*, прибавив к *x* значение *e*.

Пример параллельного программирования задачи формирования массива *B*, состоящего из ненулевых элементов массива *A*, представлен ниже [8]. Символ $ используется для обозначения номера текущего процесса.

```
intx;
x = 0;
spawn(1, n) {
int e;
e = 1;
if (A[$]! = 0) {
a = fetch_and_add(e, x)
B[a] = A[$];
}
}
```

Исходя из необходимости реализации PRAM, ЭМ должна выглядеть как "обычный" многопотоковый процессор, функционирующий под управлением операционной системы (ОС) семейства Unix.

В ЭМ под управлением ОС Minix3 могут быть "нативно" реализованы алгоритмы распараллеливания программы на множество потоков, распределения потоков, контроля загрузки ядер и т. д. При "нативном" распараллеливании фрагменты кода единой программы назначаются с очень низкими накладными расходами на множество исполнительных ядер [9]. Ядра при этом получают указатель на фрагмент кода, передаваемого им для исполнения. Результаты сохраняются в общей памяти. В относительно устоявшейся терминологии это называется подход на базе "легких потоков". Такой подход реализован, например, в библиотеке "легких тредов" Qthreads [6], созданной в Sandia National Laboraroties. Его основная идея — обеспечить механизм порождения параллельно исполняемых потоков на уровне программы, а не на уровне ОС. В идеале накладные расходы на вызов такого потока сравнимы с затратами при вызове обычной функции. В рамках формализма Qthread порождение потока описывается вызовом вида

```
void start_thread(int(*)(int), int, ...)
```

Вызов может быть из любого ядра. Исходной функции нужно передать указатель на запускаемую функцию, число аргументов и собственно аргументы, если таковые имеются. Такой вызов совместно с атомарными, не блокируемыми операциями инкремента и некоторыми другими стандартными операциями создает возможность порождать (spawn) и завершать (join) группы параллельно выполняемых потоков, создавая последовательно-параллельную программу.

Синхронизация, которая выполняется на базе примитивов ОС, как в Unix-процессах и p-тредах [10], вызывает большие задержки. В рамках предлагаемой модели вычисления, проводимые каждым процессором, представляются легким потоком — тредом. Для синхронизации тредов при этом предлагается использовать механизм межтредовой синхронизации и коммуникации на базе разделяемой памяти. Его суть в добавлении к каждому слову памяти дополнительного бита, принимающего значение full/empty (FE-бита) и использовании наряду с традиционными синхронизирующих команд обращения к памяти [11]. Команды writeef, readfe, readff и writeff, обращающиеся к ячейке памяти, могут выполняться только при определенном в них в первом компоненте суффикса значении FE-бита и оставляют после выполнения значение этого бита, заданное программистом во втором компоненте суффикса команды. Выполнение команды задерживается, если FE-бит не имеет требуемого значения. Например, команда writeff требует, чтобы перед ее выполнением значение FE-бита слова памяти, в которое будет запись, было full и оставляет после выполнения это же значение. Значение FE-бита устанавливает обычно, что ячейка памяти имеет содержимое (full) или нет (empty).

Синхронизация на базе FE-бита не требует специальных разделяемых переменных, таких как замки, семафоры и др., а также механизмов синхронизации весьма затратных по времени их определения и исполнения при миллионах тредов [6, 11]. Кроме того, применение разделяемых переменных и неделимых (атомарных) последовательностей команд, определяющих значение условия ветвления и выполняющих переход в соответствии с полученным значением, существенно сложнее и более длительно, чем выполнение команд доступа к памяти при синхронизации динамически порождаемых легких тредов. Поэтому синхронизация на базе FE-бита позволит выдавать максимально возможное в исполняемой программе число обращений к памяти, генерируемых легкими тредами, и параллельно выполнять эти доступы в расслоенной памяти.

Следует отметить, что в CrayXMT [11] расширение языка C для использования синхронизации на базе FE-битов реализовано как введение синхронизирующих переменных *x*$ и аппаратных функций (*generic functions*) для выполнения операций чтения и записи. Среди них отметим:

- purge *x*$ — присвоение FE-биту *x*$ значения empty;
- writeqr *x*$, g — запись в *x*$ значения *g*, если значение FE-бита *x*$ равно *q* или ожидание записи, пока значение FE-бита не станет *q*; после записи значение FE-бита становится равным *r*;
- readqr *x*$ — чтение значения *x*$, если значение FE-бита *x*$ равно *q* или ожидание чтения пока значение FE-бита не станет *q*; после чтения значение FE-бита становится равным *r*.

Например, фрагмент программы [11], записывающий по адресу *i* значение 2 в случае, если FE-бит равен full, и оставляющий значение этого бита не измененным, приведен ниже:

```
int i;
writeff(&i, 2);
```

Использование синхронизирующих переменных имеет свои особенности. Так, в работе [11] приводится пример условного оператора if ((*x*$ >= 10)&&(*x*$ <= 100)){}, который корректно выполняется, если программист задумал использовать два разных значения переменной *x*$ при условии, что есть другой процесс, записывающий новое значение в *x*$ после выполнения проверки *x*$ >= 10.

Если же программист хотел только проверить принадлежность *x*$ интервалу [10, 100], то это будет дедлок, избавить от которого может правильное программирование:

```
tmpx = x$;
if ((tmpx > = 10) && (tmpx < = 100)){}
```

Синхронизирующие переменные могут использоваться как семафоры для создания барьеров и критических интервалов, организующих исключительный доступ одного процесса из множества процессов к разделяемым этим множеством процессов переменным. Кроме этого, синхронизирующие переменные могут использоваться для задания потоковых (*dataflow*) вычислений. Например, вычисление $F(i, j) = 0{,}25(F(i - 1, j) + F(i - 1, j - 1) +$

+ $F(i, j-1)$ + $F(i, j)$ в области $0 < i < n + 1$, $0 < j < n + 1$ при заданных значениях $F(0, 0)$, $F(0, j)$, $F(i, 0)$ и установленных у $F(0, 0)$, $F(0, j)$, $F(i, 0)$ значениях FE-битов, равных full, может быть представлено следующим образом (символ $ используется для обозначения номера текущего процесса [8]):

```
spawn (1, n) {
int i;
i = $;
spawn (1, n) {
int j;
j = $;
purge (&(F[i, j]); // установка FE-битов, равных empty,
// в области 0<i<n + 1, 0<j<n + 1
}
}
spawn (1, n) {
int i;
i = $;
spawn (1, n) {
int j;
j = $;
double Left = readff(&(F[i, j-1]);
double BottomLeft = readff(&(F[i-1, j-1]);
double Bottom = readff(&(F[i-1, j]);
double t = readee(&(F[i, j]);
t = 0.25*(t + Left + BottomLeft + Bottom);
writeef(&(F[i, j]), t);
}
}
```

Программа порождает n тредов, каждый из которых порождает по n тредов. На первом шаге у ячеек памяти, хранящих $F(i, j)$ в области $0 < i < n + 1$, $0 < j < n + 1$, устанавливаются FE-биты, равные empty. Далее, аналогично предыдущему шагу, проводится порождение $n \times n$ тредов. В каждом треде вычисляется соответствующая функция. Сначала вычисляется $F[1, 1]$, так как исходно $F(0, 0)$, $F(0, 1)$, $F(1, 0)$ имеют FE-биты, равные full. Затем параллельно могут быть вычислены $F[2, 1]$ и $F[1, 2]$, так как появилось необходимое для их вычисления значение $F[1, 1]$ с FE-битом, равным full, затем параллельно могут быть вычислены $F[3, 1]$, $F[2, 2]$, $F[1, 3]$ и так далее.

Представляется, что рассмотренная модель параллельного программирования на базе PRAM, реализованная путем расширения языка C [8] функциями порождения, завершения асинхронных легких тредов, а также синхронизации атомарными операциями и доступа к памяти с использованием FE-битов, в достаточной мере удовлетворяет требованию "пользователь должен только указывать, какие вычисления можно проводить параллельно, сообразуясь только с выбранным алгоритмом".

Рассмотрим, как возможно реализовать функционирование в режиме потока запросов к памяти, обеспечиваемое рассмотренной моделью программирования, в условиях ограничений, обусловленных современной элементной базой.

Согласно представлению об архитектуре экзафлопсного компьютера (рис. 10), он строится из вычислительных модулей (ВМ), каждый из которых имеет один или несколько процессорных кристаллов с подсоединенными к ним блоками памяти и интерфейсами коммуникационной среды, объединяющей все вычислительные модули [7, 9, 12, 13].

Процессорный кристалл содержит накристальную коммуникационную сеть (*interconnect*), объединяющую процессорные элементы (ПЭ), состоящие из вычислительных ядер с подсоединенными к ним блоками локальной памяти, в том числе кеш-памяти, специальные вычислительные устройства, один или несколько блоков управления памятью (*Memory Management Unit* — MMU). Необходимость введения блоков памяти, создающих эффект большого процента не переключающихся в каждом такте транзисторов в занимаемой ПЭ локальной области кристалла, создает дополнительные эффекты.

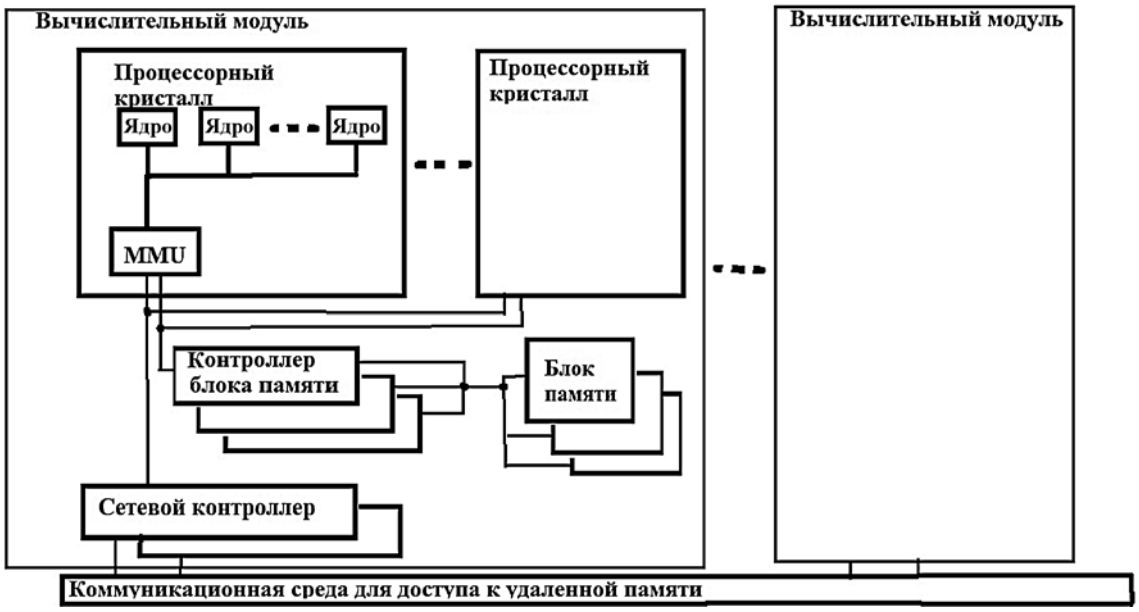

Рис. 10. Архитектура экзафлопсного суперкомпьютера

Короткие проводники передачи команд и данных между ядрами и блоками локальной памяти, к которым они обращаются, обеспечивают высокие энергоэффективности и тактовую частоту. На кристалле может разместиться много (тысячи) указанных ПЭ, функционирование которых требует загрузки и выгрузки их блоков памяти для смены программ и данных, а также инициализации выполнения в них программ. Такие архитектурные решения использованы в кристаллах разной специализации, например, PEZY-SC2 [14], ориентированном на численные методы решения дифференциальных уравнений, и Graphcore Colossus IPU [15], созданном для решения задач искусственного интеллекта. Специализация архитектуры кристалла определяет, служит ли он только полем однородных ПЭ, загружаемых и управляемых из внешнего по отношению к кристаллу хост-компьютера, как Graphcore Colossus, либо реализуется иерархическая гетерогенная кластерная архитектура, в которой на каждом уровне иерархии вычислительные устройства имеют локальные блоки памяти, как в PEZY-SC2. В этом варианте архитектуры связь между хост-компьютером и специализированным кристаллом частично перекладывается на управляющие процессоры кластеров, что снижает требования к производительности хост-компьютера и пропускной способности канала связи между ним и специализированным кристаллом или множества объединенных специализированных кристаллов.

Вычислительный модуль имеет также необходимое для обеспечения требуемой пропускной способности число смарт-контроллеров блоков распределенной разделяемой памяти, собственно блоки этой памяти и один или несколько сетевых контроллеров.

Распределенная разделяемая память суперкомпьютера состоит из совокупности блоков памяти, размещенных в каждом ВМ. Число блоков памяти в каждом ВМ, с одной стороны, должно быть как можно больше, чем обеспечивается потенциальная возможность обслуживания большего числа запросов к памяти. С другой стороны, следует иметь в виду, что их число ограничивается сложностью управления.

При инициации суперкомпьютера выполняется конфигурация глобального адресного пространства путем распределения адресного пространства по блокам памяти. Это распределение фиксируется в блоках управления памятью MMU. Каждое ядро из числа размещаемых на кристалле при выполнении текущим тредом команды доступа к разделяемой памяти по чтению или по записи после выдачи обращения к памяти приостанавливает этот тред до получения ответа от памяти. Обращение треда к памяти помещается в очередь необслуженных запросов к памяти в накристальном блоке управления памятью MMU.

На основе задаваемого при инициализации распределения глобального адресного пространства по блокам памяти блок управления памятью определяет, куда идет обращение — к локальному или удаленному блоку памяти другого ВМ. В последнем случае формируется обращение к сетевому контроллеру, который должен переслать в другой ВМ обращение соответствующему удаленному блоку памяти, поставив этот запрос в свою очередь необслуженных запросов. По получении ответа от удаленного блока памяти сетевой контроллер извлекает соответствующий запрос из очереди необслуженных и пересылает полученный ответ удаленного блока памяти в соответствующий блок управления памятью MMU.

По получении ответа от удаленного или локального блока памяти, блок управления памятью MMU извлекает соответствующий запрос из очереди необслуженных запросов к памяти и передает ядру, выдавшему запрос, результат обращения к памяти, делая возможным продолжение исполнения приостановленного треда.

Контроллер памяти для каждого запроса независимо может работать в обычном режиме (запросы на чтение и запись выполняются в порядке поступления, не используют дополнительные признаки) или в расширенном режиме с учетом механизма FE-битов. Такой бит сопровождает в памяти каждое слово.

Таким образом, выполнение в ядре команды обращения к памяти задерживается в ожидании ответа о завершении обращения к памяти из MMU. Смарт-контроллер блока памяти, получив обращение из MMU непосредственно, либо через сетевой контроллер, выполняет работу в обычном или в расширенном режиме с FE-битом указанной ячейки памяти. Смарт-контроллер блока памяти содержит память FE-битов слов памяти блоков памяти, которыми управляет этот контроллер. Если FE-бит не имеет требуемого значения, то обращение к памяти помещается в таблицу ожидания. Строки этой таблицы содержат собственно обращение к памяти (команду чтения/записи, адрес ячейки, значения FE-битов до выполнения обращения и по его завершении, служебную информацию для работы в расширенном режиме, а также указатель на незавершенную команду MMU). Изменение состояния FE-бита слова инициирует обработку таблицы ожидания с реализацией принятого подхода к разрешению конфликтов доступа к одной ячейке разделяемой памяти CREW с возможностью чтения одной и той же ячейки совокупностью тредов, а записи только одним из тредов. Причем число чтений передается из операции порождения тредов в служебной информации, и только при его достижении меняется состояние FE-бита слова. После обработки таблицы ожидания из нее выбирается обращение к памяти, которое удаляется из таблицы и запускается на выполнение.

Если FE-бит имеет требуемое значение, то контроллер блока памяти выполняет требуемое чтение или запись и формирует заданное значение FE-бита, если достигнуто заданное в служебной информации число чтений. Не вдаваясь в детали реализации контроллера памяти отметим, что после завершения обращения к ячейке памяти должен выполняться поиск в таблице ожидания строк, соответствующих этой ячейке памяти, и проверка возможности выполнения соответствующего обращения к памяти. Если таковое возможно, то обращение пускается на выполнение и соответствующая ему строка удаляется из таблицы. Естественно, описанные действия требуют ассоциативного поиска в таблице ожидания.

Распределенность обращений в память служит гарантией высокой производительности, пропорцио-

нальной числу используемых смарт-контроллеров блоков памяти вычислительных модулей.

Отметим отличие рассмотренного подхода от традиционной организации потоковых (*dataflow*) архитектур [16]. В традиционных архитектурах основу составляет ассоциативная память, в которую помещаются команды, ждущие готовности операндов. При готовности всех операндов команда извлекается из ассоциативной памяти и исполняется. Ясно, что число команд, одновременно извлекаемых из ассоциативной памяти, определяет производительность машины. Поэтому объем ассоциативной памяти должен быть большим. Но созданию такой ассоциативной памяти препятствуют затраты энергии и падение быстродействия при росте объема памяти.

Попытки программной реализации ассоциативной памяти на базе адресуемой памяти с использованием хеш-функций, разбиение общей ассоциативной памяти на локальные блоки и определение готовности только начальных команд целых фрагментов команд кардинально не изменили ситуацию [17, 18].

Однако подход на уровне контроллеров блоков памяти, выполняющих работу с FE-битами слов, представляется вполне реализуемым. Во-первых, блоков памяти может быть много, и во-вторых, выполняется только действительно необходимая синхронизация без какого-либо предварительного разбиения программ на фрагменты команд.

Реализация в процессоре внеочередного исполнения команд посредством резервирующей станции и использование контроллеров блоков памяти, выполняющих работу с FE-битами слов, покрывают всю функциональность традиционных dataflow-архитектур.

Заключение

Исследование и разработка языков и средств параллельного программирования требуют анализа существующих архитектурных идей, направленных на повышение производительности, создания экспериментальных суперкомпьютеров и программирования для них актуальных задач, исполнение которых на современных средствах неудовлетворительно по производительности и масштабируемости. Решение следует искать с разных сторон, меняя представление параллельных программ, способы отображения программных компонентов на ресурсы, вводя реализации синхронных и асинхронных тредов, программно-аппаратную поддержку синхронизации потоков и межпотоковых коммуникаций в целях определения удовлетворительного варианта архитектуры, ОС, Runtime-системы, средств подготовки программ и их компиляции. Иными словами, совместной разработки аппаратных и программных средств.

Развитие параллельного программирования возможно только в рамках конкретной модели, учитывающей "родовые" особенности (*intrisic*) архитектуры, обусловившей появление этой модели. Так для модели с передачей сообщений такой особенностью служит атомарность передач сообщений с возможностью продолжения исполнения только по завершении приема всего сообщения. В случае модели с разделяемой памятью, особенность заключается в реализации управления доступом к разделяемой ячейке памяти. Поэтому параллельное программирование для этих моделей строится на разных постулатах.

Наступил момент смены парадигмы суперкомпьютерных вычислений вообще и парадигмы программирования в частности. В этом отношении налицо сходство с аналогичной ситуацией начала 1990 гг., приведшей к замене векторно-конвейерных вычислений на базе парадигмы последовательного программирования, расширенной векторными операциями, на парадигму массово-параллельных вычислений на базе передачи сообщений.

Новая парадигма заключается в представлении всего возможного параллелизма обработки. Пользователь должен только указывать, какие вычисления можно проводить параллельными потоками над общей разделяемой памятью, сообразуясь только с выбранным алгоритмом. При необходимости прочитать одно и то же значение многими потоками, а потом записать вместо него в соответствующую ячейку общей памяти новое значение пользователь полагает, что механизм разрешения конфликта реализуется аппаратными средствами управления доступом к памяти.

Список литературы

1. **Евреинов Э. В., Косарев Ю. Г.** Однородные универсальные вычислительные системы высокой производительности. Новосибирск: Наука, 1966. 308 с.
2. **Фортов В. Е., Левин В. К., Савин Г. И.** и др. Суперкомпьютер МВС-1000М и перспективы его применения // Наука и промышленность России. 2001. № 11. С. 49—52.
3. **Корнеев В. В.** Архитектура вычислительных систем с программируемой структурой. Новосибирск: Наука, 1985. 166 с.
4. **Besta M., Hoefler T.** Slim Fly: A Cost Effective Low-Diameter Network Topology. USA // SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. New Orleans, Louisiana, 16-21 Nov. 2014. P. 348—359.
5. **Deng Y., Guo M., Ramos A. F.** et al. Optimal Low-Latency Network Topologies for Cluster Performance Enhancement. arXiv:1904.00513v1 [cs.NI].
6. **Wheeler K., Murphy R., Thain D.** Qthreads: An API for Programming with Millions of Lightweight Threads // 2008 IEEE International Symposium on Parallel and Distributed Processing, 14-18 April 2008. URL: https://ieeexplore.ieee.org/document/4536359
7. **Корнеев В. В.** Подход к программированию суперкомпьютеров на базе многоядерных мультитредовых кристаллов // Вычислительные методы и программирование. 2009. Том 10. С. 123—128.
8. **Wen X., Vishkin U.** FPGA-Based Prototype of a PRAM-On-Chip Processor // CF '08 Proceedings of the 5th conference on Computing frontiers ACM. New York, 2008. P. 55—66.
9. **Елизаров С. Г., Лукьянченко Г. А., Корнеев В. В.** Технология параллельного программирования экзафлопсных компьютеров // Программная инженерия. 2015. № 7. С. 3—10.
10. **Institute** of Electrical and Electronics Engineers. IEEE Std 1003.1-1990: Portable Operating Systems Interface (POSIX.1), 1990. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/fipspub151-2.pdf
11. **Kopser A., Vollrath D.** Overview of the Next Generation Cray XMT // 53rd Cray User Group meeting, CUG 2011. Fairbanks, Alaska. 2011. P. 1-10. URL: https://cug.org/5-publications/proceedings_attendee_lists/CUG11CD/pages/1-program/final_program/Monday/03C-Kopser-Paper.pdf
12. **Корнеев В. В.** Модель программирования — смена парадигмы // Открытые системы. 2010. № 3. С. 29—31.
13. **Корнеев В. В.** Модель программирования и архитектура экзафлопсного суперкомпьютера // Открытые системы. 2014. № 10. С. 20—22.
14. **Torii S., Ishikawa H.** ZettaScaler: Liquid immersion cooling Manycore based Supercomputer // ISC 2017. Frankfurt am Main,

Germany, June 18—22, 2017. URL: https://www.pezy.co.jp/wp-content/uploads/2019/02/Keynote_candar2017.pdf
15. **Jia Zh., Tillman B., Maggioni M., Scarpazza D.** Dissecting the Graphcore IPU Architecture via Microbenchmarking. Technical Report. December 7, 2019. arXiv:1912.03413v1 [cs.DC] 7 Dec 2019.
16. **Gurd J., Bohm W., Teo Y.** Performance Issues in Dataflow Machines // Elsevier Science Publishers (North-Holland) Future Generations Computer Systems 3. 1987. P. 285—297.
17. **Бурцев В. С.** Выбор новой системы организации выполнения высокопараллельных вычислительных процессов, примеры возможных архитектурных решений построения суперЭВМ // Параллелизм вычислительных процессов и развитие архитектуры СуперЭВМ. М.: ИВВС РАН, 1997. С.41—78
18. **Климов А. В., Левченко Н. Н., Окунев А. С.** Преимущества потоковой модели вычислений в условиях неоднородных сетей. // Информационные технологии и вычислительные системы. 2012. № 2. С. 36—45.

Parallel Programming

V. V. Korneev, korv@rdi-kvant.ru, Research and Development Institute "Kvant", Moscow, 125438, Russian Federation

Corresponding author:

Korneev Victor V., Principal Researcher, Research and Development Institute "Kvant", Moscow, 125438, Russian Federation
E-mail: korv@rdi-kvant.ru

Received on November 11, 2021
Accepted on November 22, 2021

As a source for the introduction of parallel programming models, the article substantiates the development of the technology of VLSI and the corresponding change in the architecture of computing systems in the direction of increasing the parallelism of processing. That is, if the architecture changes significantly, then the parallel programming model should change in order to reduce the difficulty of effectively mapping program to hardware resources. A model of parallel programming on shared memory and synchronization based on FE-bits of shared memory words is considered. The architecture of the computing system of the exaflops performance level is also proposed, for which the considered programming model is adequate. The article tries to convey to the reader that this architecture originated as a synthesis of the development of VLSI technology, architecture of parallel systems and high-production parallel programming. It is indicated that when implementing the proposed architecture and programming model, an architecture with data flow processing is implemented. A parallel programming model for computing systems with local connections and optimal graphs of machine-to-machine connections are presented.

Keywords: *VLSI, architecture, parallel programming model, data flow processing, interconnection graph.*

For citation:

Korneev V. V. Parallel Programming**,** *Programmnaya Ingeneria*, 2022, vol. 13, no. 1, pp. 3—16.

DOI: 10.17587/prin.13.3-16

References

1. **Evreinov E. V., Kosarev Y. G.** *Homogeneous universal high-performance computing systems*, Novosibirsk, Nauka, 1966, 308 p. (in Russian).
2. **Fortov V. E., Levin V. K., Savin G. I., Zabrodin A. V., Karatanov V. V., Elizarov G. S., Korneev V. V., Shabanov B. M.** The MVS-1000M supercomputer and its application prospects. *Nauka i promyshlennost Rossii*, 2001, vol. 55, no. 11, pp. 49—52 (in Russian).
3. **Korneev V. V.** *Architecture of computer systems with a programmable structure*, Novosibirsk, Nauka, 1985, 166 p. (in Russian).
4. **Besta M., Hoefler T.** Slim Fly: A Cost Effective Low-Diameter Network Topology, *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis,* New Orleans, Louisiana, 16—21 Nov. 2014, pp. 348—359.
5. **Deng Y., Guo M., Ramos A. F., Huang X., Xu Zh., Liu W.** Optimal Low-Latency Network Topologies for Cluster Performance Enhancement. arXiv:1904.00513v1 [cs.NI].
6. **Wheeler K., Murphy R., Thain D.** Qthreads: An API for Programming with Millions of Lightweight Threads, *Workshop on Multithreaded Architectures and Applications at IEEE IPDPS*, April, 2008, available at: https://ieeexplore.ieee.org/document/4536359
7. **Korneev V. V.** An approach to programming supercomputers based on multicore multithreaded VLSI chips, *Computational Methods and Programming*, 2009, vol. 10, pp. 123—128 (in Russian).
8. **Wen X., Vishkin U.** FPGA-Based Prototype of a PRAM-On-Chip Processor, *CF '08 Proceedings of the 5th conference on Computing frontier, s* ACM New York, 2008, pp. 55—66.
9. **Elizarov S. G., Lukyanchenko G. A., Korneev V. V.** Technology of parallel programming of exaflops computers, *Programmnaya ingeneria,* 2015, no. 7, pp. 3—10 (in Russian).
10. **Institute** of Electrical and Electronics Engineers. IEEE Std 1003.1-1990: Portable Operating Systems Interface (POSIX.1), 1990, available at: https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/fipspub151-2.pdf
11. **Kopser A., Vollrath D.** Overview of the Next Generation Cray XMT, *53rd Cray User Group meeting, CUG 2011*, Fairbanks, Alaska, 2011, pp. 1—10, available at: https://cug.org/5-publications/proceedings_attendee_lists/CUG11CD/pages/1-program/final_program/Monday/03C-Kopser-Paper.pdf
12. **Korneev V. V.** Programming model — paradigm shift, *Otrrytye sistemy,* 2010, no. 3, pp. 29—31 (in Russian).
13. **Korneev V. V.** Programming model and architecture of an exaflop supercomputer, *Otrrytye sistemy,* 2014, no. 10, pp. 20—22 (in Russian).
14. **Torii S., Ishikawa H.** ZettaScaler: Liquid immersion cooling Manycore based Supercomputer, *ISC 2017*, Frankfurt am Main, Germany, June 18—22, 2017, available at: https://www.pezy.co.jp/wp-content/uploads/2019/02/Keynote_candar2017.pdf
15. **Jia Zh., Tillman B., Maggioni M., Scarpazza D.** Dissecting the Graphcore IPU Architecture via Microbenchmarking. Technical Report. December 7, 2019. arXiv:1912.03413v1 [cs.DC] 7 Dec 2019.
16. **Gurd J., Bohm W., Teo Y.** Performance Issues in Dataflow Machines, *Elsevier Science Publishers (North-Holland) Future Generations Computer Systems 3,* 1987, pp. 285—297.
17. **Burtsev V. S.** The choice of a new system for organizing the execution of highly parallel computing processes, examples of possible architectural solutions for building supercomputers, *Parallelism of computing processes and the development of the architecture of supercomputers*, Moscow, IVVS RAN, 1997, pp. 41—78. (in Russian).
18. **Klimov A. V., Levchenko N. N., Okunev A. S.** Advantages of a data flow computing model in heterogeneous networks, *Information technologies and computing systems*, 2012, no. 2, pp. 36—45 (in Russian).