УДК 681.3.06

ПРОГРАММИРОВАНИЕ НА ТИПОВЫХ АЛГОРИТМИЧЕСКИХ СТРУКТУРАХ С МАССИВНЫМ ПАРАЛЛЕЛИЗМОМ

П. К. Берзигияров[1]

Рассматривается новая технология программирования, основанная на ограниченных формах параллелизма — типовых алгоритмических структурах с массивным параллелизмом. Приведена классификация типовых алгоритмических структур. На основе анализа задач вычислительной газодинамики и квантовой химии предложены новые проблемно-ориентированные структуры. Рассмотрены различные механизмы композиции таких структур, применяемые в процессе разработки параллельных программ, а также вопросы их оптимизации. Описаны особенности программной реализации этих структур, включая механизмы обеспечения мобильности и эффективности, а также объектно-ориентированные методы повторного использования и специализации типовых алгоритмических структур. Представлена архитектура системы программирования.

Ключевые слова: параллельное программирование, типовые алгоритмические структуры, оптимизация программ, массивный параллелизм

1. Введение. Технологические аспекты разработки эффективных вычислительных программ с массивным параллелизмом связаны с решением целого комплекса проблем, в частности, масштабируемостью алгоритмов и программ, мобильностью программ и разработкой систематических методов и технологий повторного использования уже готовых программных компонентов.

Одним из ключевых моментов при проведении крупномасштабных вычислительных экспериментов является сокращение сроков разработки параллельных приложений. Предлагаемый в настоящей работе подход базируется на использовании ограниченных форм параллелизма, представляющих собой Типовые (повторяющиеся) Алгоритмические Структуры с массивным параллелизмом, и на разработке механизмов их повторного использования (ТАС) [1, 7]. В программировании эти структуры известны под самыми различными названиями: алгоритмические шаблоны (skeletons, templates) [13, 14, 18, 30, 31], родовые алгоритмы (generic algorithms), архетипы (archetypes) [26], типовые проектные решения (design patterns) [21–25, 34, 35].

Систематические исследования типовых алгоритмических структур и на их основе методов программирования были начаты в рамках функционального программирования работой М. Коула [14]. В результате усилий различных групп были введены и исследованы такие алгоритмические структуры, как MAP (применить-ко-всем), $REDUCE$ (редукция), $SCAN$, $PIPE$ (конвейер), $FARM$, $DIVIDE\text{-}AND\text{-}CONQUER$ (разделяй-и-властвуй) и др., которые представлялись в виде параметрических функций высших порядков [1, 4, 5, 7, 10–14, 16, 18–20, 28, 29–31, 37, 38, 42, 44, 45]. Кроме того, были исследованы различные способы иерархической и горизонтальной композиции этих структур с целью разработки сложных программ из более простых [12, 30, 31, 42, 44, 45]. Известен целый ряд экспериментальных систем, таких как SPF [19], $SkelML$ [12], $SKIL$ [10], $SKIPPER$ [33], основанных на использовании функциональных языков.

Другое направление, тесно связанное с первым, развивается в рамках процедурных языков программирования, в частности, $P3L$ [30, 31] и $SKIE$ [5]. Эти работы основаны на расширении последовательных языков программирования конструкциями для задания алгоритмических шаблонов.

Наконец, последнее направление развивается в рамках объектно-ориентированного подхода и базируется на использовании типовых проектных решений (design patterns) [15]. Основной акцент в этих работах делается на разработку развитых механизмов повторного использования и специализации типовых структур. Известно несколько экспериментальных систем этого класса, в частности, $FrameWorks$ [41], $DPnDP$ [34, 35], $Enterprise$ [25, 41], CO_2P_3S [21–24].

В настоящей работе также рассматривается подход к разработке параллельных программ, основанный на использовании типовых алгоритмических структур с массивным параллелизмом. В разделе 2

[1] Институт проблем химической физики РАН, Институтский просп., 18, 142432, Московская обл., Ногинский район, п. Черноголовка; e-mail: parvaz@pro.icp.ac.ru

рассматриваются универсальные алгоритмические структуры, возникающие естественным образом практически в каждой содержательной задаче. В разделе 3 на основе анализа задач вычислительной газодинамики и квантовой химии предложено несколько проблемно-ориентированных типовых алгоритмических структур с параллелизмом по данным. В разделе 4 подробно рассмотрены различные механизмы композиции таких структур, применяемые в процессе разработки параллельных программ. В разделе 5 в качестве примера рассмотрено представление CIP-метода [2] посредством типовых алгоритмических структур. В разделе 6 рассмотрены вопросы оптимизации ТАС. Наконец, в разделе 7 описаны особенности программной реализации этих структур, включая механизмы обеспечения мобильности и эффективности, а также объектно-ориентированные методы повторного использования и специализации типовых алгоритмических структур.

2. Базовые алгоритмические структуры. В настоящей работе под типовой алгоритмической структурой мы понимаем семейство однотипных параметризованных алгоритмов

$$S = A(P_S, P_F), \tag{2.1}$$

где S — множество алгоритмов A с набором P_S структурных и P_F функциональных параметров.

Подход предполагает реализацию типовых алгоритмических структур в виде шаблонов-заготовок, в которых зафиксирована инвариантная часть, представляющая распределенную коммуникационную структуру. Настройка шаблона на конкретную область применения осуществляется фиксацией структурных и функциональных параметров. Структурные параметры определяют топологию параллельной программы, в то время как функциональная часть определяет вычислительное содержание алгоритма.

Множество типовых алгоритмических структур можно разбить на *базовые* и *проблемно-ориентированные* структуры. Базовые алгоритмические структуры не ориентированы на конкретную область применения и носят универсальный характер. Проблемно-ориентированные алгоритмические структуры — это высокоуровневые конструкции, характеризующиеся более узкой областью применимости и возникающие в результате обобщения группы близких вычислительных методов, например, многосеточных методов [11, 29] или методов частиц [20].

Рассмотрим вначале базовые алгоритмические структуры. Примерами таких структур могут служить $PIPE$ (конвейер), DAC (разделяй-и-властвуй), $REDUCE$ (редукция), MAP (применить-ко-всем), $FARM$ и др. [1, 7, 8, 13, 18, 28, 30, 31, 37, 38]. В настоящей работе мы ограничимся структурами, ориентированными на параллелизм по данным (data-parallel) [30, 31, 37, 38]. Опишем эти структуры подробнее. Для удобства мы зафиксируем некоторый формат описания типовых алгоритмических структур, который включает в себя следующие компоненты: наименование, назначение, применимость, формальное описание, графическое представление, параметры.

Для пояснения композиционной структуры алгоритмических шаблонов и рассмотрения их основных компонентов мы будем использовать диаграммы следующего вида:

$$\begin{array}{ccc} X & \overset{TAS(f_i)}{\longrightarrow} & Y \\ d_X \downarrow & & \downarrow c_Y \\ X_i & \overset{f_i}{\longrightarrow} & Y_i \end{array}$$

где $X = X_1 \times X_2 \times \cdots \times X_n$ и $Y = Y_1 \times Y_2 \times \ldots \times Y_n$ — представляют собой входной набор данных и результат, $Tas(f_i)$ — типовая алгоритмическая структура, f_i $(1 \leqslant i \leqslant n)$ — функции-параметры структуры, реализующие собственно параллельную часть вычислений

$$\begin{array}{c} X_1 \xrightarrow{f_1} Y_1, \\ X_2 \xrightarrow{f_2} Y_2, \\ \cdots \\ X_n \xrightarrow{f_n} Y_n, \end{array}$$

где d_X — функции-проекторы, осуществляющие декомпозицию исходной монолитной структуры данных на распределенные фрагменты, c_Y — функция, осуществляющая сборку частичных (распределенных) результатов снова к монолитному виду.

Таким образом, исходная типовая алгоритмическая структура представляет собой последовательную композицию

$$TAS(f) = c_Y \circ f_i \circ d_X, \quad 1 \leqslant i \leqslant n, \tag{2.2}$$

следующих трех функций

$$\begin{aligned} X &\xrightarrow{d_X} X_i, \\ X_i &\xrightarrow{f_i} Y_i, \\ &\dots \\ Y_i &\xrightarrow{c_Y} Y. \end{aligned} \tag{2.3}$$

Отметим, что подобное представление рассматривалось неформально на уровне программной реализации в работах [30, 31]. Более абстрактная форма этого представления использовалась в работе [16] к представлению отдельных функций в рамках BMF-формализма [8], а также в [28] с целью исследования композиционной структуры алгоритмов "разделяй-и-властвуй". Наконец, в работе [9] такая форма использовалась на самом высоком абстрактном уровне (операторов на банаховых пространствах) с целью декомпозиции параллельных вычислений на собственно параллельную часть f_i $(1 \leqslant i \leqslant n)$ и на так называемую "перпендикулярную" часть, представляемую проекторами d_X и c_Y, служащими для распределения исходных данных и сборки результата.

Такой подход к математическому описанию типовых алгоритмических структур позволяет учитывать не только параллельную часть структуры, но и декомпозицию исходных данных и сборку результатов, что существенно расширяет набор допустимых оптимизирующих преобразований ТАС и их композиций.

Применить-ко-всем (MAP). Эта структура описывает применение некоторой функции (процедуры) ко всем элементам распределенной структуры данных (списка, массива и т.д) [8]. Представляет наиболее фундаментальную форму параллелизма — независимый параллелизм по данным, или параллелизм в "чистом" виде. Присутствует практически во всех алгоритмах (очень часто не как самостоятельная, а как вспомогательная при описании более содержательных структур). В композиционном плане эта структура может быть представлена диаграммами на рис. 2.1.

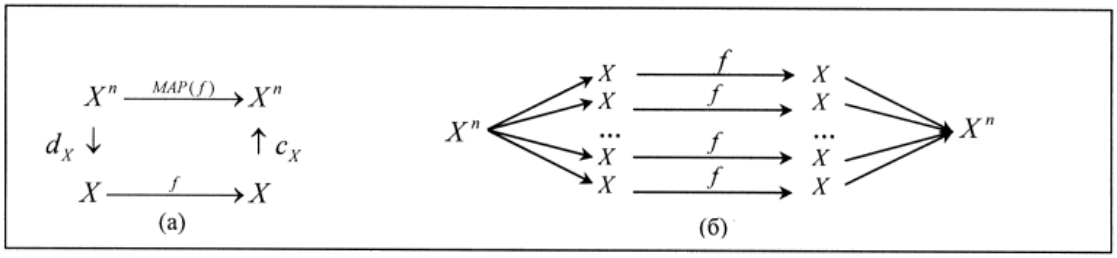


Рис. 2.1. MAP-структура

Структурные параметры: топологические параметры структур данных (размерности массивов в нашем случае равны n).

Функциональные параметры: функция f, применяемая ко всем элементам структуры данных.

Редукция ($REDUCE$). Эта структура представляет параллельную редукцию для ассоциативной бинарной операции (рис. 2.2). Широко применяется практически во всех параллельных численных алгоритмах. Ввиду своей фундаментальности введена в состав библиотеки MPI [17, 40]. Очень часто используется как отдельная функция сборки результатов.

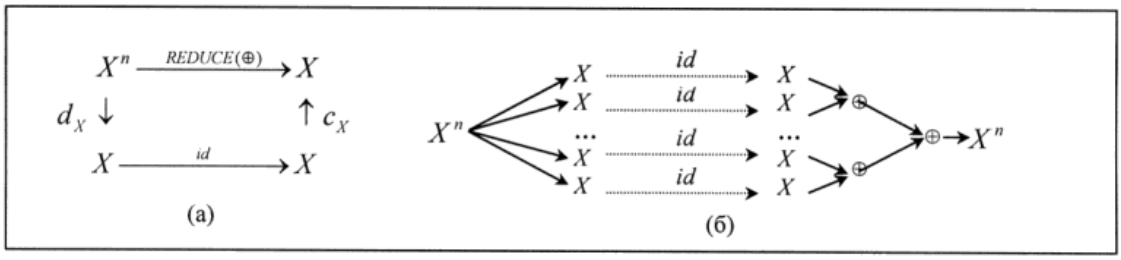


Рис. 2.2. $REDUCE$-структура

Структурные параметры: топологические параметры структур данных (размерности массивов равны n)

Функциональные параметры: $\oplus$ — бинарная ассоциативная и коммутативная операция ($+$, $*$, min, max и т.д.).

Разделяй-и-властвуй (DAC). Осуществляет рекурсивную декомпозицию исходной задачи на подзадачи вплоть до возможности непосредственного решения, с последующей комбинацией частичных ре-

зультатов в конечный результат (рис. 2.3). Характеризуется широкой областью применимости (линейная алгебра, сортировка и т.д.).

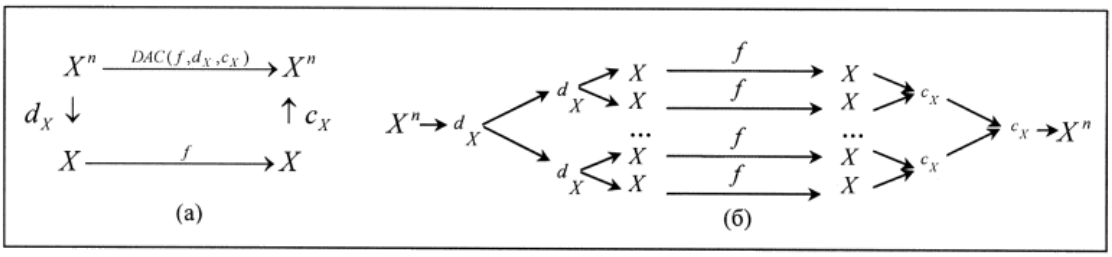


Рис. 2.3. *DAC*-структура

Структурные параметры: топологические параметры структуры (размер входных данных — n).

Функциональные параметры: функция f построения решения для примитивной задачи применяется к структурам данных, размерность которых позволяет получить решение непосредственно, например, с помощью правила Крамера для решения систем линейных уравнений размерности 2 и 3, d_X и c_Y — функции декомпозиции и сборки соответственно.

Дублирование вычислений ($REPLICATE$). Ни один достаточно содержательный алгоритм не может не содержать последовательных вычислений. Обычно, последовательные вычисления выполняются одним процессом, а затем результат рассылается остальным процессам, участвующим в работе. В ряде случаев, более выгодной является схема вычислений и соответствующая ей алгоритмическая структура, основанные на дублировании последовательных вычислений с целью экономии на обменах. Как видно из приведенной ниже диаграммы (рис. 2.4), в этой структуре отсутствуют затраты на декомпозицию и сборку данных (соответствующие проекторы представлены тождественными функциями), а последовательная функция совпадает с параллельной.

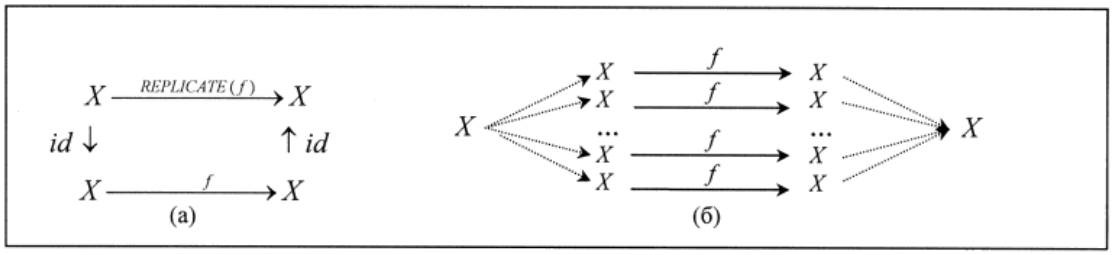


Рис. 2.4. $REPLICATE$-структура

Структурные параметры: отсутствуют.

Функциональные параметры: последовательная функция f.

Последовательные вычисления (SEQ). Процесс декомпозиции алгоритма на параллельные, относительно независимые фрагменты не может продолжаться бесконечно. Рано или поздно мы будем вынуждены прекратить декомпозицию. Оставшиеся неделимые, или атомарные, части алгоритма будут представлять собой последовательные модули, выступающие в качестве строительных блоков будущей программы. В иерархической древовидной структуре программы этим модулям будут соответствовать листья. Для представления последовательных модулей программы и будет использоваться типовая алгоритмическая структура SEQ.

На этом в целях экономии места мы ограничим рассмотрение базовых структур. Отметим только, что существуют и другие часто применяемые структуры с параллелизмом по данным, в частности, $SCAN$ и ZIP, а также структуры $PIPE$ и $FARM$, ориентированные на функциональный параллелизм (task-parallelism) и обработку потоков данных (streams) [4, 30, 31].

3. Проблемно-ориентированные алгоритмические структуры. В настоящем разделе мы введем ряд наиболее распространенных алгоритмических структур, ориентированных на задачи вычислительной газодинамики и квантовой химии. Следует отметить, что большинство из них не ограничивается только этими областями, а на самом деле имеет более широкую область применимости. С другой стороны, существуют и другие структуры (в частности, ТАС), которые ориентированы на многосеточные методы [11, 29] (такие структуры в данной работе не обсуждаются). Таким образом, рассматриваемый здесь набор проблемно-ориентированных ТАС отражает те численные методы, с которыми автор сталкивался в своей практике. Заметим только, что наш подход позволяет по мере необходимости пополнять каталог ТАС.

Вычисления на регулярных сетках ($MESH$). Параллельное применение некоторой функции (процедуры) ко всем узлам регулярной сетки. В процессе вычислений используется информация из соседних узлов, набор которых зависит от применяемого шаблона (рис. 3.1). Эта типовая структура имеет несколько разновидностей. Во-первых, по числу измерений различают одно- двух- и трехмерные сетки. Кроме того, мы будем различать итерационные и эволюционные сетки. В первом случае ТАС реализует итерационный цикл с проверкой на каждом шаге некоторого условия сходимости. Во втором случае задается временной интервал, на котором осуществляется моделирование поведения системы с некоторым шагом. Таким образом, в первом случае выход из ТАС осуществится только по завершении внутреннего циклического (итерационного) процесса. Во втором случае структура используется в комбинации с другими шаблонами в рамках некоторого внешнего по отношению к ТАС цикла. Следует отметить, что второй случай легко сводится к первому заданием истинного условия сходимости.

Рассмотрим наиболее распространенные на практике алгоритмические структуры, возникающие при реализации разностных методов решения уравнений математической физики.

Данные. Представление данных базируется на N-мерной сетке (массиве), обычно $N = 1, 2, 3$. С каждым узлом сетки может быть связано несколько переменных. Заметим, что 1- и 2-мерные сетки являются частными случаями 3-мерной, поэтому нет необходимости для каждого измерения вводить свою типовую структуру. Могут присутствовать также и величины, не связанные с конкретным узлом (как правило, это — глобальные константы).

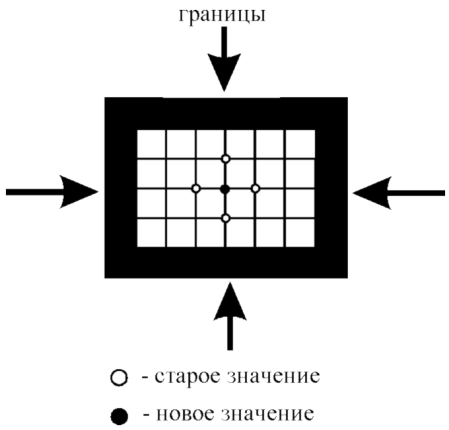


Рис. 3.1. Сеточные вычисления

Вычисления состоят в общем случае из следующей последовательности операций.

Декомпозиция и распределение данных. Для многомерных сеток применяют различные стратегии: построчную, поколоночную, блочную (рис. 3.2). При этом различают простую декомпозицию и циклическую. Глобальные константы, как правило, дублируются. Стратегия декомпозиции зависит от системы уравнений и является предметом оптимизации.

Вычисления для каждого узла одной или нескольких переменных с использованием значений из ближайших узлов.

Обмены с нелокальными (граничными) узлами, участвующими в вычислениях новых значений для узлов данного процесса. На рис. 3.2 представлена схема взаимодействия для двумерной поблочной декомпозиции. С целью оптимизации коммуникаций к каждому блоку добавляются фиктивные элементы для хранения нелокальных границ своих соседей, участвующие в вычислениях данного процесса (рис. 3.1).

Проверка на завершение (тест на сходимость для итерационных сеток). Очень часто на этом этапе выполняются операции глобальной редукции над всей сеткой (поиск максимума или минимума, суммирование и др.). Для эволюционных сеток этот этап отсутствует.

В композиционном плане структура описывается диаграммой на рис. 3.3. Мы будем различать в своих дальнейших построениях итерационные ($MESHI$) и эволюционные сетки ($MESH$).

Структурные параметры. Топологические параметры сетки (в нашем случае n).

Функциональные параметры. Функция f, применяемая ко всем узлам сетки.

Попарные взаимодействия ($CROSS$). Это наиболее распространенная структура в вычислительной квантовой химии, связанная с вычислением тех или иных величин для пар взаимодействующих атомов либо орбиталей. В частности, по такой схеме реализуются вычисления элементов матрицы Фока, построение основного гамильтониана и т.д. [3]. В общем случае нам необходимо вычислить попарные взаимодействия всех компонентов системы, состоящей из n частиц. При этом вычисление взаимодействий для пар, сформированных различными частицами, отличается от вычислений для пар, сформированных одной и той же частицей.

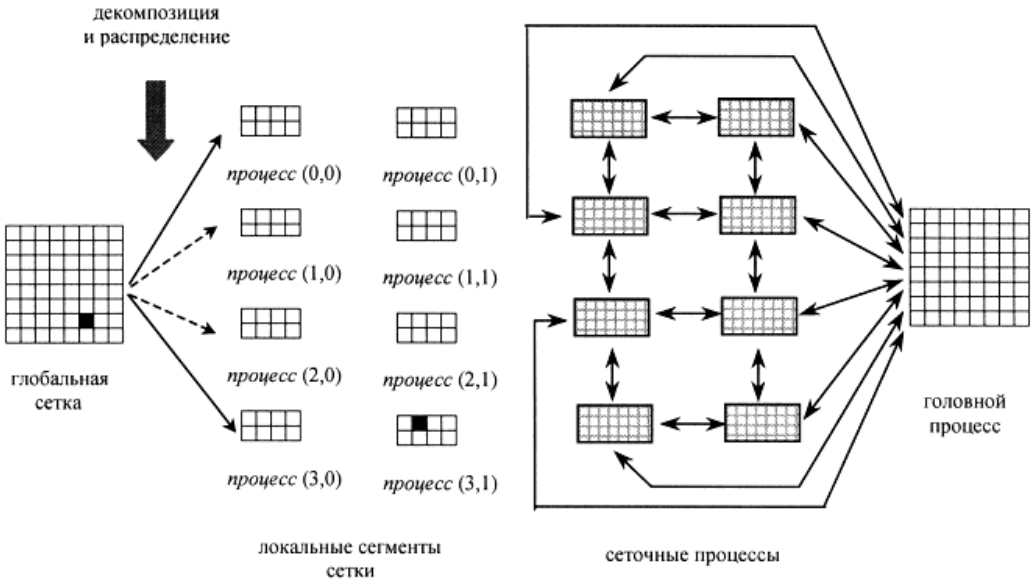


Рис. 3.2. Декомпозиция данных и межпроцессные взаимодействия

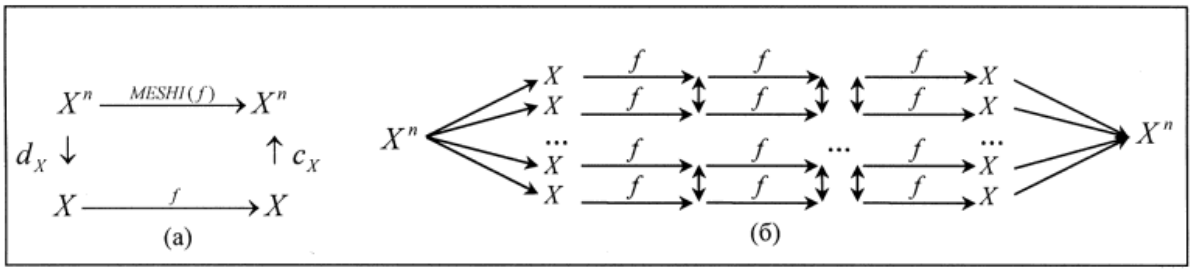


Рис. 3.3. $MESH$-структура

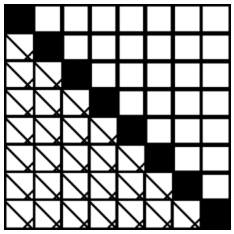


Рис. 3.4. Попарные взаимодействия в системе из восьми частиц

Эту задачу легко представить в матричном виде. На рис. 3.4 приведен пример для системы, состоящей из восьми частиц. Как видно из рисунка, при вычислениях осуществляется обход нижнего треугольника матрицы размерности $n \times n$, где n — количество взаимодействующих частиц. Вычисления всех элементов взаимно независимы. При этом вычисление диагональных элементов отличается от вычислений элементов нижнего треугольника (рис. 3.4, 3.5). Таким образом, каждый из этих фрагментов может быть представлен с помощью типовой алгоритмической структуры MAP. Так же, как и для сеток, различают итерационный и эволюционный случаи.

Структурные параметры. n — количество взаимодействующих частиц.

Функциональные параметры. f_d — функция, используемая для вычисления диагональных элементов; $f_\triangle$ — функция, используемая для вычисления элементов нижнего треугольника.

4. Управляющие алгоритмические структуры: композиция и вложенность. Рассмотрим различные механизмы композиции типовых алгоритмических структур, применяемых в процессе разработки параллельных программ.

4.1. Композиция последовательных модулей с помощью единичных ТАС. Простейшим способом построения параллельных программ является композиция последовательных модулей посредством единичных ТАС. В этом случае глобальная структура программы полностью определяется применяемой алгоритмической структурой. В иерархическом плане программа представляет собой двухуровневое дерево, в корне которого находится ТАС, а листья дерева образованы последовательными модулями-

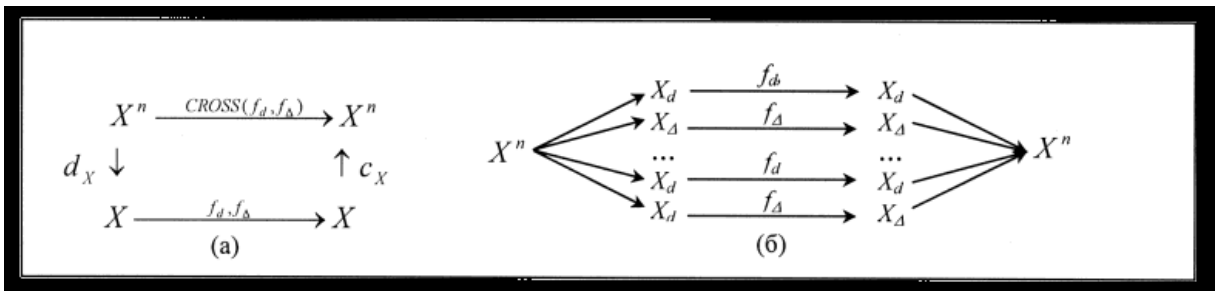


Рис. 3.5. *CROSS*-структура

параметрами ТАС.

Рассмотрим в качестве примера задачу анимации изображений, обрабатывающую последовательность графических изображений, называемых кадрами. Исходную задачу можно представить в виде последовательности трех подзадач:

Generate: вычисляет положение и перемещения каждого объекта, формирующего кадр;

Geometry: выполняет вырезку, проецирование и другие преобразования;

Display: удаляет невидимые поверхности и антиальясинг, а затем сохраняет кадр на диске; далее считывается следующий кадр, и процесс обработки повторяется.

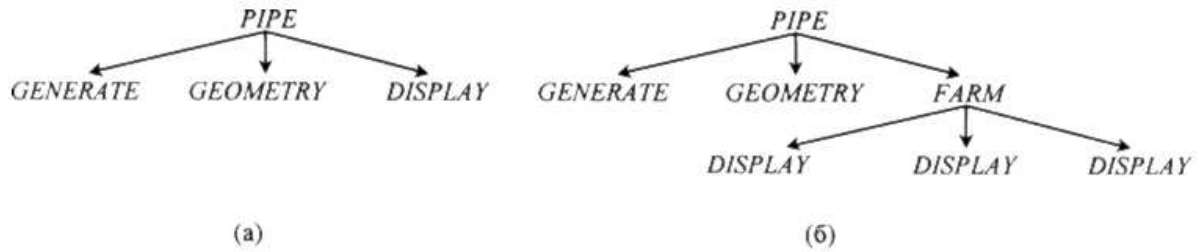


Рис. 4.1. Композиция ТАС

При обработке множества кадров наиболее естественной ТАС для реализации алгоритма является конвейер, состоящий из трех стадий, каждая из которых реализует одну из подзадач. Программа имеет двухуровневую иерархическую структуру, корнем которой является ТАС $PIPE$, а листья сформированы последовательными модулями *Generate*, *Geometry* и *Display* (рис. 4.1 а). Каждая из стадий получает на вход кадр, выполняет свою часть вычислений и передает обработанный кадр на следующую стадию.

Таким образом, в простейшем случае ТАС выступают в качестве операторов композиции или конструкторов, позволяющих собирать сложные программы из последовательных модулей. К сожалению, использование ТАС в качестве единственного способа композиции накладывает слишком сильные ограничения на структуру программы.

4.2. Иерархическая композиция ТАС: вложенность. Если снять ограничение, состоящее в том, что функциональными параметрами ТАС могут быть только последовательные модули, и разрешить использовать другие ТАС в качестве строительных блоков, то мы сможем строить программы с вложенной иерархической структурой. Иерархическая структура программы представляет собой дерево, в корне которого находится ТАС, определяющая глобальную структуру программы, нетерминальные вершины образованы ТАС, а листья дерева, как и ранее, соответствуют последовательным модулям.

Рассмотрим вновь задачу анимации изображений. Как правило, наиболее трудоемкая фаза обработки выполняется последовательным модулем *Display*, поэтому мы можем осуществить декомпозицию этого модуля посредством ТАС $FARM$, который запустит несколько копий последовательного модуля *Display*, ускорив тем самым обработку в целом (рис. 4.1 б).

Вложенность идеально соответствует нисходящему методу разработки программ “сверху-вниз”, позволяя проводить иерархическую декомпозицию исходной задачи на отдельные подзадачи.

4.3. Горизонтальная композиция алгоритмических структур. Программы, полученные с помощью уже рассмотренных механизмов композиции, имеют одну и ту же общую особенность. В иерархическом представлении глобальная структура программы и все ее нетерминальные вершины соответствуют единичным ТАС. В силу того, что все ТАС, за исключением последовательных, имеют единую регулярную структуру с массивным параллелизмом, то неявно предполагается, что задача на любом уровне иерархии, кроме самых нижних, имеет свою единую структуру, представленную той или иной ТАС. Очевидно, что это выполняется далеко не всегда.

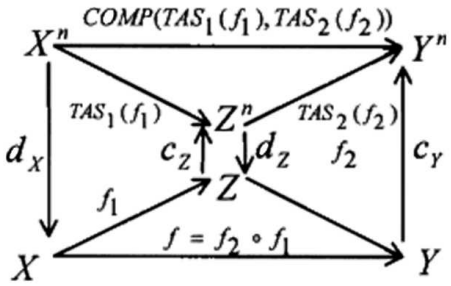
Во-первых, любая нетривиальная задача содержит последовательные фрагменты. Во-вторых, даже

на одном и том же уровне иерархии программа может содержать различные алгоритмические структуры. В-третьих, на практике наряду с нисходящими методами разработки так же широко применяют и восходящие методы “снизу-вверх”. Для всех этих целей требуются дополнительные механизмы композиции, которые позволяли бы осуществлять представление программ в виде линейной композиции ТАС. Для этих целей мы введем управляющие ТАС.

Последовательная композиция ($COMP$). Рассмотрим теперь последовательную композицию ТАС. В частности, диаграмма для композиции двух ТАС, у которых совпадают область значений первой и область определения второй алгоритмической структуры

$$\begin{aligned} &TAS_1(f_1) : X^n \to Z^n, \\ &TAS_2(f_2) : Z^n \to Y^n, \end{aligned} \tag{4.1}$$

выглядит следующим образом:



Функциональные параметры. $TAS_1(f_1)$, $TAS_2(f_2)$ — типовые алгоритмические структуры, для которых выполняется последовательная композиция $COMP\big(TAS_1(f_1), TAS_2(f_2)\big)$.

Цикл ($LOOP$). Тело цикла, состоящее из вложенной ТАС, исполняется требуемое число раз. Различают циклы параметрические (или for-циклы) и итерационные (while-циклы с проверкой условия в начале и repeat-циклы с проверкой условия в конце).

Функциональные параметры. $TAS(f)$ — типовая алгоритмическая структура, составляющая тело цикла; $cond$ — условие выхода из цикла.

5. Идентификация ТАС в CIP-методе. Представление CIP-метода с помощью ТАС. Анализ структуры CIP-метода показывает [2, 43], что в нем содержится представительный набор типовых алгоритмических структур и он достаточно легко и естественно реализуется на их базе.

Рассмотрим отдельные этапы алгоритма с указанием конкретных алгоритмических структур, соответствующих им, в предположении, что внешний цикл представляется шаблоном $LOOP$:

— вычисление шага в подобласти реализуется шаблоном MAP,

— определение минимального из шагов соответствует шаблону $REDUCE$,

— далее следует рассылка остальным процессам,

— затем идет последовательность вычислений различных величин на сетке, которые представляются шаблонами $MESH$.

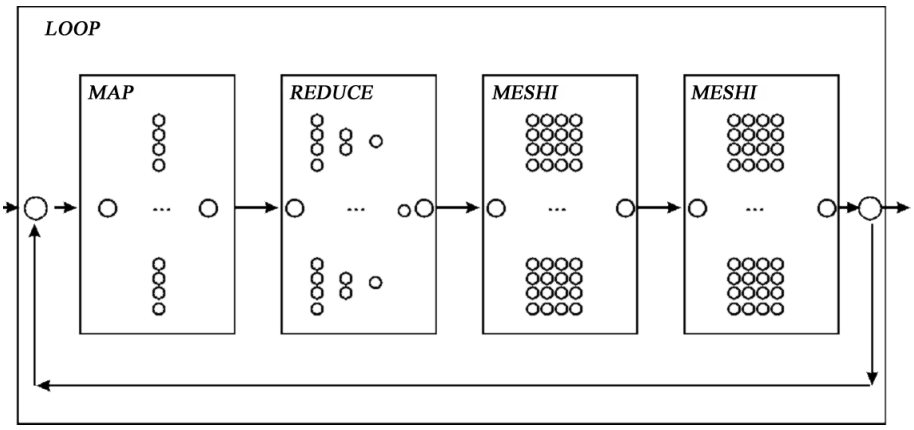


Рис. 5.1. Представление CIP-метода посредством ТАС

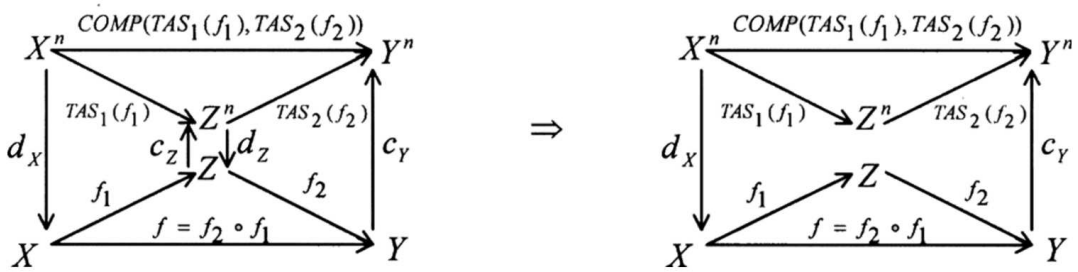
Таким образом, в данном алгоритме идентифицировано четыре шаблона, в частности: $LOOP$, MAP, $REDUCE$ и $MESH$ (несколько вхождений). На рис. 5.1 представлена структура программы с использованием указанных шаблонов.

6. Оптимизирующие преобразования ТАС. Рассмотрим некоторые вопросы оптимизации типовых алгоритмических структур.

6.1. Оптимизация горизонтальных композиций ТАС. Основная проблема при реализации управляющих конструкций заключается в оптимизации интерфейса между алгоритмическими структурами, над которыми выполняется горизонтальная композиция. Задача состоит в исключении ненужных этапов сборки и декомпозиции промежуточных результатов между ними. Поясним эту проблему на примере последовательной композиции двух алгоритмических структур

$$COMP\big(TAS_1(f_1), TAS_2(f_2)\big) \tag{6.1}$$

Пусть, как и ранее, область значений первой структуры совпадает с областью определения второй и для проекторов выполняется следующее условие: $c_Z = d_Z^{-1}$. Тогда, принимая во внимание, что $d_Z \circ c_Z = id$, получим



$$COMP\big(TAS_1(f_1), TAS_2(f_2)\big) = TAS_2(f_2) \circ TAS_1(f_1) = c_y \circ f_2 \circ d_z \circ c_z \circ f_1 \circ d_x = c_y \circ f_2 \circ f_1 \circ d_x. \tag{6.2}$$

Для этого случая легко выписать и доказать набор оптимизирующих правил преобразований, устраняющих избыточные функций рассылки и сборки промежуточных результатов. Рассмотрим вначале правила, в которых результирующая функция $c_y \circ f_2 \circ f_1 \circ d_x$ может быть представлена одной из ТАС, являющихся параметрами композиции $COMP$:

$$COMP\big(MAP(d_1, f, c_1), MAP(d_2, g, c_2)\big) \to MAP(d_1, g \circ f, c_2),$$
$$COMP\big(MAP(d_1, f, c_1), REDUCE(d_2, \oplus)\big) \to REDUCE(d_1, \otimes), \quad x_1 \otimes x_2 = f(x_1) \oplus f(x_2),$$
$$COMP\big(MAP(d_1, f, c_1), DAC(d_2, p, g, c_2)\big) \to DAC(d_1, p, g \circ f, c_2),$$
$$COMP\big(MAP(d_1, f, c_1), MESH(d_2, g, I, c_2)\big) \to MESH(d_1, g \circ f, I, c_2),$$
$$COMP\big(MAP(d_1, f, c_1), CROSS(d_2, p, g_1, g_2, c_2)\big) \to CROSS(d_1, p, g_1 \circ f, g_2 \circ f, c_2),$$
$$COMP\big(DAC(d_1, p, f, c_1), DAC(d_2, p, g, c_2)\big) \to DAC(d_1, p, g \circ f, c_2),$$
$$COMP\big(DAC(d_1, p, f, c_1), MAP(d_2, g, c_2)\big) \to DAC(d_1, p, g \circ f, c_2),$$
$$COMP\big(MESH(d_1, f, I, c_1), MESH(d_2, g, I, c_2)\big) \to MESH(d_1, g \circ f, I, c_2),$$
$$COMP\big(MESH(d_1, f, I, c_1), MAP(d_2, g, c_2)\big) \to MESH(d_1, g \circ f, I, c_2),$$
$$COMP\big(CROSS(d_1, p, f_1, f_2, c_1), CROSS(d_2, p, g_1, g_2, c_2)\big) \to CROSS(d_1, p, g_1 \circ f_1, g_2 \circ f_2, c_2),$$
$$COMP\big(CROSS(d_1, p, f_1, f_2, c_1), MAP(d_2, g, c_2)\big) \to CROSS(d_1, p, g \circ f_1, g \circ f_2, c_2).$$

Рассмотрим оптимизирующие правила для случая, когда результирующая функция $c_y \circ f_2 \circ f_1 \circ d_x$ не может быть представлена ни одной из ТАС, являющихся параметрами композиции $COMP$. Однако и в этом случае правые части выражений оказываются проще за счет исключения ненужных этапов сборки

и декомпозиции:

$$COMP\big(MAP(d_1,f,c_1),MESHI(d_2,p,g,I,c_2)\big)\to COMP\big(MAP(d_1,f,id),MESHI(id,p,g,I,c_2)\big),$$
$$COMP\big(DAC(d_1,p,f,c_1),REDUCE(d_2,\oplus)\big)\to COMP\big(DAC(d_1,p,f,id),REDUCE(id,\oplus)\big),$$
$$COMP\big(DAC(d_1,p,f,c_1),MESH(d_2,g,I,c_2)\big)\to COMP\big(DAC(d_1,p,f,id),MESH(id,g,I,c_2)\big),$$
$$COMP\big(DAC(d_1,p,f,c_1),MESHI(d_2,p,g,I,c_2)\big)\to COMP\big(DAC(d_1,p,f,id),MESHI(id,p,g,I,c_2)\big),$$
$$COMP\big(MESH(d_1,f,I,c_1),REDUCE(d_2,\oplus)\big)\to COMP\big(MESH(d_1,f,I,id),REDUCE(id,\oplus)\big),$$
$$COMP\big(MESH(d_1,f,I,c_1),DAC(d_2,p,g,c_2)\big)\to COMP\big(MESH(d_1,f,I,id),DAC(id,p,g,c_2)\big),$$
$$COMP\big(MESH(d_1,f,I,c_1),MESHI(d_2,p,g,I,c_2)\big)\to COMP\big(MESH(d_1,f,I,id),MESHI(id,p,g,I,c_2)\big),$$
$$COMP\big(MESHI(d_1,p,f,I,c_1),MAP(d_2,g,c_2)\big)\to COMP\big(MESHI(d_1,p,f,I,id),MAP(id,g,c_2)\big),$$
$$COMP\big(MESHI(d_1,p,f,I,c_1),REDUCE(d_2,\oplus)\big)\to COMP\big(MESHI(d_1,p,f,I,id),REDUCE(id,\oplus)\big),$$
$$COMP\big(MESHI(d_1,p,f,I,c_1),DAC(d_2,p,g,c_2)\big)\to COMP\big(MESHI(d_1,p,f,I,id),DAC(id,p,g,c_2)\big).$$

Наконец, в общем случае, когда область значений первой и область определения второй структуры не совпадают, а частично пересекаются, и когда для функций декомпозиции исходных данных и сборки результата соотношение $d_Z\circ c_Z=id$ в целом не выполняется, мы будем применять более общее правило, состоящее в исключении взаимно-обратных подфункций, входящих в состав d_Z и c_Z.

Рассмотрим более подробно функции d_Z и c_Z. В общем случае операция декомпозиции может быть представлена в виде [4, 30, 31, 45]

$$d_x=bcast\circ scatter\circ multicast. \tag{6.3}$$

Здесь

— функция $bcast(x)$ рассылает x всем процессам; например, при поэлементном умножении вектора на скаляр $y_i=a*x_i$ $(1\leqslant i\leqslant n)$ функция $bcast(a)$ рассылает a всем процессам, участвующим в операции;

— функция $scatter(x)$ осуществляет декомпозицию структуры x (например, на сегменты для одномерного массива или на блоки для матрицы) и рассылает их по одному на каждый из участвующих в вычислениях процессов;

— функция $multicast(x)$ осуществляет разбиение структуры x (например, на сегменты для одномерного массива или на блоки для матрицы) и рассылает каждый из них некоторому подмножеству процессов, участвующих в вычислениях.

Что касается сборки результатов, то для ее осуществления в общем случае применяют следующую композицию:

$$c_x=gather\circ reduce. \tag{6.4}$$

Здесь

— функция $gather(x)$ осуществляет сборку фрагментов структуры x, размещая их последовательно на процессе-коллекторе;

— функция $reduce(x)$ совмещает выполнение операции и сборку результата.

Применительно к этому набору функций возможны следующие оптимизирующие операции:

— исключение дублирующей декомпозиции (в частности, широковещательной рассылки); применяется в том случае, если требуется разослать величину, которая уже была разослана ранее и не модифицировалась в процессе обработки, т.е. когда отсутствует вхождение этой переменной в набор выходных параметров предыдущей ТАС;

— исключение вхождений взаимно-обратных подфункций.

В нашем случае в роли взаимно-обратных функций выступают $scater(x)$ и $gather(x)$, для которых справедливо

$$scater(x)\circ gather(x)=id. \tag{6.5}$$

В общем случае, мы имеем следующие правила оптимизации

$$COMP\big(TAS_1(f_1)\,[d_z(x)],TAS_2(f_2)\,[d_z(x)]\big)\to COMP\big(TAS_1(f_1),TAS_2(f_2)\,[d_z(x)\leftarrow id]\big), \tag{6.6}$$

$$COMP\big(TAS_1(f_1)\,[d_z],TAS_2(f_2)\,[c_z]\big)\to COMP\big(TAS_1(f_1)\,[d_z\leftarrow id],TAS_2(f_2)\,[c_z(x)\leftarrow id]\big), \tag{6.7}$$

где $[s\leftarrow r]$ обозначает выражение, в котором выполнена подстановка подвыражения r вместо всех вхождений выражения s.

Для функций $bcast(x)$, $scatter(x)$ и $gather(x)$ эти правила примут вид

$$COMP\big(TAS_1(f_1)\,[bcast(x)], TAS_2(f_2)\,[bcast(x)]\big) \to COMP\big(TAS_1(f_1), TAS_2(f_2)\,[bcast(x) \leftarrow id]\big),$$
$$COMP\big(TAS_1(f_1)\,[scatter], TAS_2(f_2)\,[gather]\big) \to COMP\big(TAS_1(f_1)\,[scatter \leftarrow id], TAS_2(f_2)\,[gather \leftarrow id]\big).$$

6.2. Оптимизация вложенных композиций ТАС. Эффективность вложенных ТАС напрямую лимитируется трудностями, связанными с оптимизацией совместного применения множества ТАС. Поэтому вопросы оптимизации вложенных алгоритмических структур заслуживают отдельного и детального рассмотрения. В частности, в работах других авторов [4, 12, 18, 19, 30, 31, 39, 42, 44, 45] рассмотрены существующие подходы к построению иерархических ТАС и методам их оптимизации. В настоящей работе мы ограничимся списком только некоторых из возможных правил преобразований:

$$MAP\big(d_1, MAP(d_2, f, c_2), c_1\big) \to MAP(d_2 * \circ d_1, f, c_1 * \circ c_2),$$
$$MAP\big(d_1, DAC(d_2, p, f, c_2), c_1\big) \to DAC(d_2 * \circ d_1, p, f, c_1 * \circ c_2),$$
$$MAP\big(d_1, MESH(d_2, f, I, c_2), c_1\big) \to MESH(d_2 * \circ d_1, f, I, c_1 * \circ c_2),$$
$$MAP\big(d_1, MESHI(d_2, p, f, I, c_2), c_1\big) \to MESHI(d_2 * \circ d_1, p, f, I, c_1 * \circ c_2),$$
$$MAP\big(d_1, CROSS(d_2, p, f_1, f_2, c_2), c_1\big) \to CROSS(d_2 * \circ d_1, p, f_1, f_2, c_1 * \circ c_2),$$
$$MAP\big(d_1, CROSSI(d_2, p_c, p, f_1, f_2, c_2), c_1\big) \to CROSSI(d_2 * \circ d_1, p_c, p, f_1, f_2, c_1 * \circ c_2),$$

где $f * [x_1, x_2, \dots x_n] = \big[f(x_1), f(x_2), \dots f(x_n)\big]$.

7. Особенности реализации типовых алгоритмических структур. Рассмотрим некоторые вопросы программной реализации этих структур, включая механизмы обеспечения мобильности и эффективности, а также объектно-ориентированные методы их повторного использования и специализации.

7.1. Модель параллельной программы. Параллельная программа, состоящая из ТАС, представляет собой сеть взаимодействующих процессов. В общем случае для каждой типовой алгоритмической структуры имеется процесс-эмиттер E, распределяющий данные в соответствии с функцией d_X , процесс-коллектор C, собирающий результаты в соответствии с функцией c_X, и множество процессов W, реализующих содержательную часть алгоритма в параллельном режиме в соответствии с функцией f_i, $1 \leqslant i \leqslant n$ (рис. 7.1).

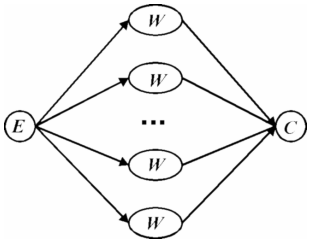


Рис. 7.1. Модель ТАС

7.2. Промежуточная абстрактная машина: обеспечение мобильности и эффективности ТАС. Важным моментом при разработке параллельных вычислительных программ является обеспечение их мобильности, т.е., переносимости программ на различные программно-аппаратные платформы. Традиционно под мобильностью программ понимают переносимость только программного кода. Вместе с тем, существует и другой аспект мобильности, связанный с эффективностью. Для учета этого аспекта предлагается рассматривать программу вместе с оценкой ее эффективности. Мы будем считать программу

$$P = P(Code, Cost) \tag{7.1}$$

мобильной, если мобильными являются ее программная реализация $Code$ и оценки эффективности $Cost$. Под мобильностью эффективности мы понимаем инвариантность аналитического выражения, оценивающего ее. Механизмы обеспечения мобильности параллельного программного кода хорошо разработаны и базируются на использовании стандартных коммуникационных библиотек типа MPI [17, 40] и последовательных языков программирования. Что касается эффективности, то одним из возможных подходов является использование промежуточного представления программ на основе абстрактных параллельных моделей [36, 37, 46].

Абстрактная BSP машина [36, 46]. Представляет собой распределенную систему (рис. 7.2 а), состоящую из:

— набора процессорных элементов, каждый со своей локальной памятью,

— коммуникационной сети,

— устройства синхронизации работы процессоров.

BSP-компьютер представляет собой абстракцию параллельной машины, которая состоит из набора четырех параметров: s — производительность процессоров, g — проницаемость сети для непрерывного потока сообщений в равновероятные пункты назначения, l — стоимость барьерной синхронизации, p — количество доступных процессоров. Вычисления в BSP (рис. 7.2 б) — это последовательность супершагов, состоящих из трех фаз:

— локальных вычислений,

— коммуникаций,

— барьерной синхронизации.

Модель обеспечивает простую оценку для вычисления стоимости супершага

$$T_{\text{sstep}} = w + h * g + l, \tag{7.2}$$

где w — есть максимальная из работ, выполняемых процессорами на этапе локальных вычислений, h — максимальное количество отправляемых (принимаемых) процессорами сообщений на фазе коммуникации (параметры g и l нормируются по отношению к производительности процессоров). Преимущество этой модели состоит в возможности параметрического описания практически всех известных параллельных архитектур [36]. Переход от одной параллельной системы к другой осуществляется простой сменой количественных значений этих параметров, которые определяются экспериментально на основе программного тестирования компьютера. Оценки эффективности программы и типовых алгоритмических структур выражаются через эти же параметры; таким образом, их аналитическое выражение не изменяется при переходе на другую систему. Общая схема системы программирования с использованием абстрактной BSP-машины представлена на рис. 7.6.

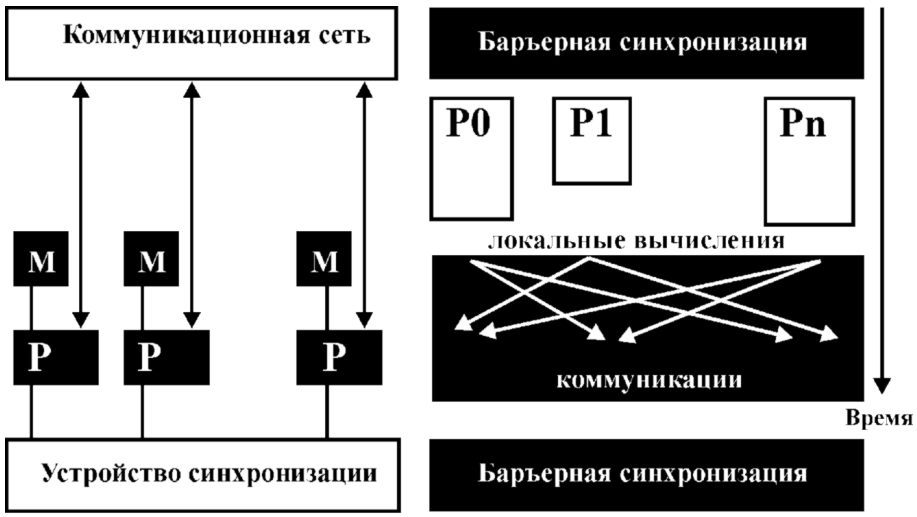


Рис. 7.2. BSP-компьютер (а) и супершаг BSP-вычислений (б)

7.3. Объектная ориентированность: механизмы повторного использования и специализации ТАС. Программирование на базе типовых алгоритмических структур предполагает разработку удобных механизмов их повторного использования. Известно несколько подходов к реализации типовых алгоритмических структур:

— функциональный подход, в рамках которого алгоритмические структуры представляются параметрическими функциями высших порядков [1, 4, 5, 7, 10–14, 16, 18–20, 28, 29–31, 37, 38, 42, 44, 45]; при всей своей теоретической привлекательности этот подход не может служить адекватной основой для программирования крупномасштабных вычислительных задач (главным образом, из-за низкой эффективности конечных программ и несоответствия языковых средств текущей вычислительной практике);

— реализация в виде библиотек подпрограмм [42]; основным препятствием для применения этого подхода является невозможность проведения оптимизации композиций;

— непосредственное расширение языка конструкциями, реализующими типовые алгоритмические структуры [5, 30, 31]; недостатком этого подхода является закрытость системы и необходимость использования нового языка;

— объектно-ориентированные методы [21–25, 34, 35, 41]; эти методы позволяют строить открытые и расширяемые системы без привлечения новых языковых средств и содержат развитые методы и технологии повторного использования.

В пионерской работе [15] рассмотрена новая объектно-ориентированная технология, основанная на идентификации типовых проектных решений (design patterns). В самом общем виде они представляют собой абстрактные (метапрограммные) описания универсальных методов, выявленных в результате анализа программных разработок из самых разных областей и идентификации повторяющихся схем. Всего в работе описано 23 типовых проектных решения, ориентированных, к сожалению, на последовательное программирование.

Нетрудно заметить близость этого подхода и подхода, основанного на применении ТАС. Дело в том, что ТАС как объекты повторного использования позволяют зафиксировать инвариантную часть алгоритмической структуры, в которой сосредоточены параллельные вычисления и коммуникации (декомпозиция–параллельные вычисления–сборка результатов). Специфика приложения определяется позже пользователем системы. Ему необходимо только задать требуемые параметры и определить модули, осуществляющие последовательную обработку (листья в иерархическом представлении программы).

К сожалению, как уже отмечалось выше, типовые проектные решения в том виде, в котором они предложены в работе [15], ориентированы на последовательное программирование. Поэтому большой практический интерес представляет интеграция этих двух подходов. Первые попытки такой интеграции были предприняты в работах [21–25, 34, 35, 41], которые завершились разработкой нескольких экспериментальных систем: $FrameWorks$ [41], $DPnDP$ [34, 35], $Enterprise$ [25, 41], CO_2P_3S [21–24]. Основой для такой интеграции служит интерпретация ТАС в качестве типовых решений, ориентированных в отличие от известных типовых проектных решений на параллельные системы.

Многоуровневая модель повторного использования. Одним из главных преимуществ объектно-ориентированного подхода и технологии программирования на базе типовых проектных решений заключается в наличии развитых механизмов повторного использования различных программных объектов. Поэтому мы остановимся на этих вопросах подробнее. Рассмотрим вначале многоуровневую модель повторного использования, которая базируется на иерархии программных компонентов, выступающих в качестве объектов повторного использования. Мы остановимся всего на двух наиболее интересных для нас уровнях иерархии.

Повторное использование "в малом": библиотеки подпрограмм и классов. Самый низкий уровень в иерархии объектов повторного использования формируется в традиционных языках программирования библиотеками подпрограмм, а в объектно-ориентированном случае — библиотеками классов. В свою очередь, этот уровень можно разбить на несколько подуровней. Так например, набор классов, обеспечивающих обработку базовых структур данных (массивов, списков, деревьев) формируют самый нижний подуровень, в то время как библиотеки математических подпрограмм (такие как ScaLAPACK) формируют самый верхний подуровень. Объектом повторного использования выступает не вся библиотека, а только отдельные ее компоненты. Использование библиотечных подпрограмм и классов основано на знании их внешнего интерфейса. Программирование же общей структуры приложения и компонентов, не покрываемых библиотекой, остается за программистом.

Повторное использование "в большом". Объектом повторного использования на этом уровне выступает не отдельная библиотечная подпрограмма или класс, а набор взаимосвязанных классов, обеспечивающий решение некоторого множества близких задач. В англоязычной литературе эти системы обозначаются термином *frameworks* [15, 21–23]. Ввиду отсутствия хорошего русского аналога этого термина, мы в дальнейшем будем использовать название "программная система-каркас" или "просто каркасная система". Каркасная система определяет архитектуру приложения, его общую структуру, разбиение на классы и объекты, фиксирует схемы взаимодействия классов и объектов, а также потоки управления. Каркасная система фиксирует проектные решения, приемлемые и инвариантные для всей предметной области, на которую она ориентирована. Специфика приложения определяется позже пользователем системы. Можно сказать, что в этом случае речь идет не о простом использовании программного кода (классов и объектов), а о повторном использовании проектных решений и микроархитектуры системы. Повторное использование на этом уровне есть инверсия по отношению к предыдущему случаю. На первом уровне наследовались отдельные компоненты системы, основа же системы, ее микроархитектура разрабатывалась программистом; в этом случае наследуется основа системы, а отдельные компоненты используются для настройки системы на конкретное применение и функциональность. Упрощая, можно сказать, что в первом случае программист пишет тело главной программы, а модули использует готовые; во втором случае тело главной программы уже готово, а необходимо написать модули. Примерами каркасных систем служат $POOMA$ [32] и $PETS$ [6, 27].

Библиотека классов. На рис. 7.4 представлена библиотека классов, реализующих типовые алгорит-

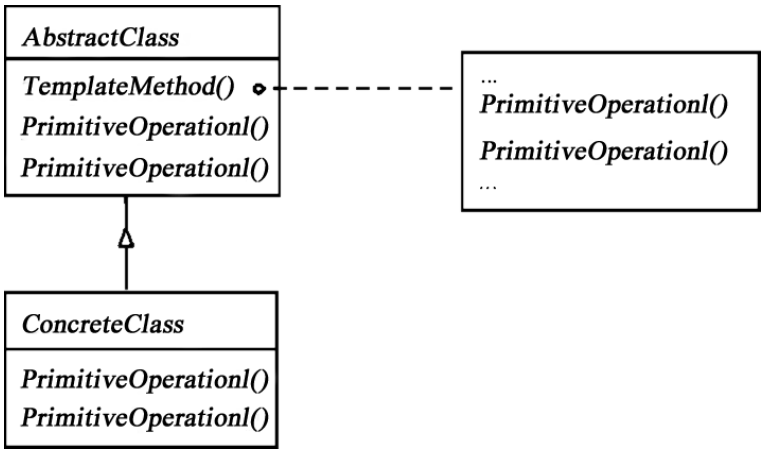


Рис. 7.3. Структура TEMPLATE METHOD

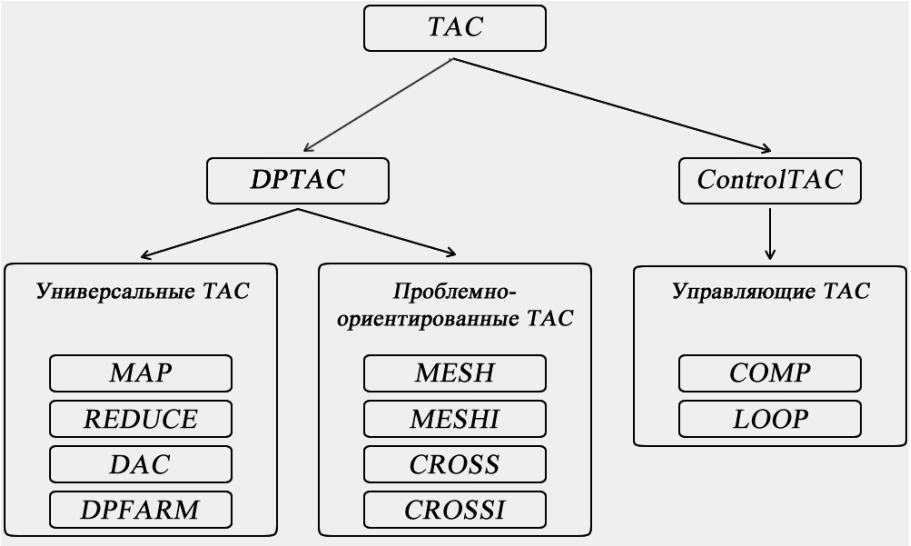


Рис. 7.4. Иерархия классов TAC

мические структуры. Как мы уже отмечали выше, с точки зрения объектно-ориентированного проектирования, TAC есть типовые проектные решения, ориентированные на параллельную обработку данных. При реализации мы опираемся на их представление в виде абстрактных классов [15]. Другим важным моментом является использование типового проектного решения TEMPLATE METHOD (рис. 7.3), как одного из наиболее удобных механизмов специализации и повторного использования TAC.

Это один из наиболее фундаментальных объектно-ориентированных методов повторного использования, который позволяет определить скелет алгоритма, оставляя детали подклассам. Он дает возможность подклассам переопределять некоторые шаги алгоритма без изменения его структуры. В нашем случае этот метод используется для реализации инвариантной части алгоритмических структур. Переменная часть, т.е. вычислительная специфика алгоритмической структуры (последовательные вычисления), реализуется через hook-операции [15, 21–24].

Архитектура системы. В заключение рассмотрим архитектуру системы (рис. 7.5). В соответствии с нашим подходом, на вход системы подается программа, представленная в виде композиции TAC.

Транслятор, используя шаблоны-заготовки, разработанные в расчете на абстрактную BSP-машину, переводит исходную программу в промежуточное BSP-представление. Далее выполняется фаза оптимизации, состоящая в частности в оптимизации интерфейса между TAC. Математическая суть этого процесса рассмотрена нами в предыдущих разделах. Наконец, на завершающей фазе осуществляется перевод оптимизированной BSP-программы в конечную программу при помощи транслятора используемой системы. Заметим, что в соответствии с нашим подходом промежуточное представление не зависит от типа используемой системы. Специализация на конкретную архитектуру осуществляется на фазе оптимизации заданием BSP-параметров. В процессе оптимизации используются параметрические оценки эффективности типовых алгоритмических структур. Эти оценки, как и программный код, не зависят от типа системы, так как их аналитическое выражение инвариантно по отношению к системе. Настройка на систему осуществляется через BSP-параметры, определяемые для каждого компьютера путем программного тестирования.

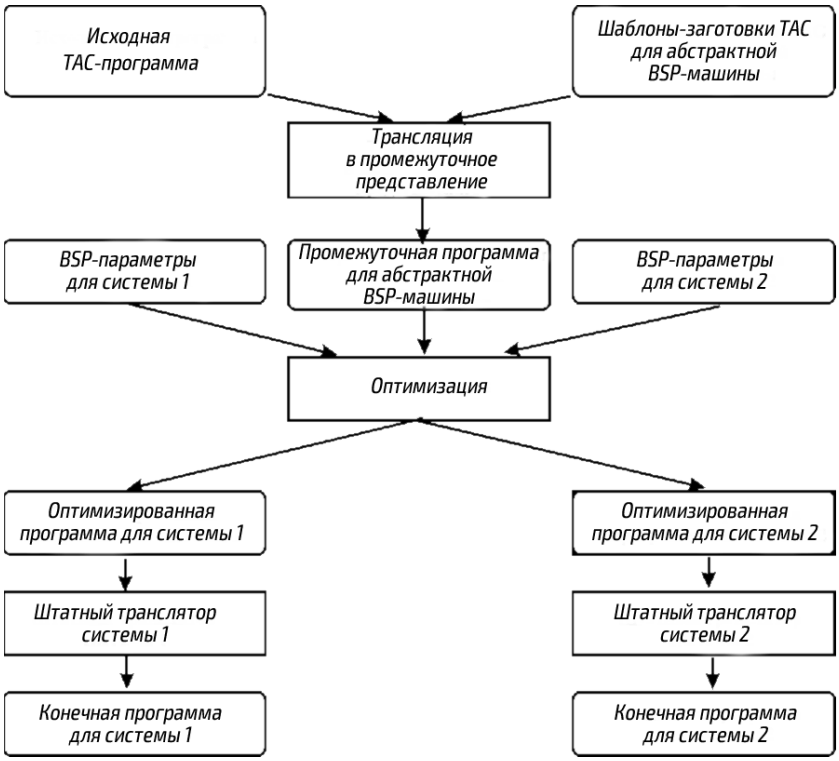


Рис. 7.5. Архитектура системы

В работе [36] приведены такие параметры для большого списка параллельных систем.

Работа выполнена при поддержке РФФИ, грант 99–07–90370.

СПИСОК ЛИТЕРАТУРЫ

1. *Берзигияров П.К.* Формальное конструирование программ на базе алгоритмических структур с массовым параллелизмом: статическая конвейеризация алгоритмов “разделяй-и-властвуй” // Тез. докл. VI-й конференции “Транспьютерные системы и их применение”. Вестник Российской транспьютерной ассоциации. Домодедово, 1996. 46.
2. *Берзигияров П.К., Султанов В.Г.* Параллельное моделирование рэлей-тейлоровских гидродинамических неустойчивостей // Вестн. Моск. ун-та. Вычисл. матем. Киберн. 2001. № 1. 1–10.
3. *Грибов Л.А., Муштакова С.П.* Квантовая химия. М.: Гардарики, 1999.
4. *Aldinucci M., Coppola M., Danalutto M.* Rewriting skeleton programs: how to evaluate the data-parallel stream-parallel tradeoff // Proc. of International Workshop on Constructive Methods for Parallel Programming. Technical Report MIP-9805. University of Passau. Passau, 1998.
5. *Bacci B., Danelutto M., Pelagatti S., Vanneschi M.* SkIE: an heterogeneous environment for HPC applications // Parallel Computing. 1999. **25**, N 13–14. 1827–1852.
6. *Balay S., Gropp W.D., McInnes L.C., Smith B.F.* Efficient management of parallelism in object-oriented numerical software libraries // Modern Software Tools in Scientific Computing / Arge E., Bruaset A.M., Langtangen H.P. (Eds.). Birkhauser Press: 1997. 163–202.
7. *Berzigyarov P.K.* Static Pipelines for Divide-and-Conquer Functions: Transformations and Analysis. Preprint IVTAN, N 8–391. Moscow, 1995.
8. *Bird R.S.* An introduction to the theory of lists // Logic of Programming and Calculi of Discrete Design / Broy M. (Ed.). NATO ASI Series. 1987.
9. *Boglaev Yu.P.* On the parallel and perpendicular computations // Parallel Algorithms and Applications. 1994. **3**. 89–107.
10. *Botorog G.H., Kuchen H.* Skil: an imperative language with algorithmic skeletons for efficient distributed programming // Proc. of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5). IEEE Computer Society. 1996. 243–252.
11. *Botorog G.H., Kuchen H.* Algorithmic skeletons for adaptive multigrid methods // LNCS. 1995. **980**. 27–41.
12. *Bratvold T.A.* Skeleton-based parallelization of functional programs: PhD thesis. Dept. of Computing and Electrical Engineering. Heriot–Watt University. Edinburgh, 1994.
13. *Campbell D.K.G.* Towards the Classification of Algorithmic Skeletons. Technical Report YCS 276. Department of Computer Science, University of York. York, 1996.
14. *Cole M.I.* Algorithmic Skeletons: Structured Management of Parallel Computation. Massachusetts, Cambridge. Boston: MIT Press, 1989.
15. *Gamma E., Helm R., Johnson R., Vlissides J.* Design Patterns: Elements of Reusable Object-Oriented Software.

Reading–Amsterdam–Tokyo: Addison–Wesley, 1995.
16. *Gorlatch S., Bischof H.* A generic MPI implementation for a data-parallel skeleton: formal derivation and application to FFT // Information Processing Letters. 1998. **8**, N 4. 447–458.
17. *Gropp W., Huss-Ledermann S., Lumsdaine A., Lusk E., Nitzberg B., Saphir W., Snir M.* MPI: The Complete Reference. Vol. 2. The MPI Extensions. Boston: MIT Press, 1998.
18. *Darlington J., Field A.J., Harrison P.G., Kelly P.H.J., Sharp D.W.N., Wu Q., While R.L.* Parallel programming using skeleton functions // In: Parallel Architectures and Languages. Berlin: Springer–Verlag, 1993. 146–160.
19. *Darlington J., Guo Y., To H.W., Yang J.* SPF: Structured parallel Fortran // Proc. of the Sixth Parallel Computing Workshop. Japan. Kawasaki, 1996. 1–6.
20. *Decyk V. K.* Skeleton PIC Codes for Parallel Computers. Physics Department. UCLA. Los Angeles, 1994.
21. *MacDonald S., Szafron D., Schaeffer J., Bromling S.* Generating parallel program frameworks from parallel design patterns // Proc. of EuroPar'2000. Berlin: Springer-Verlag, 2000. 95–104.
22. *MacDonald S., Szafron D., Schaeffer J., Bromling S.* From patterns to frameworks to parallel programs // IEEE Concurrency. 1999. **7**, N 1–4. 1–21.
23. *MacDonald S., Szafron D., Schaeffer J.* Object-oriented pattern-based parallel programming with automatically generated frameworks // Proc. of the Fifth USENIX Conference on Object-Oriented Tools and Systems. San Diego, 1999. 29–43.
24. *MacDonald S.* Parallel Object-Oriented Pattern Catalogue. Draft. 1998.
25. *MacDonald S.* Design patterns in enterprise // Proc. of CASCON'96. Toronto, 1996. 1–10.
26. *Massingill B.L., Chandy K.M.* Parallel Program Archetypes. Technical Report CS-TR-96-28. California Institute of Technology. Pasadena, 1996.
27. *McInnes L.C., Smith B.F.* PETSc 2.0: A case study of using MPI to develop numerical software libraries // Proc of the 1995 MPI Developers Conference. University of Notre Dame. Notre Dame, 1995. 1–16.
28. *Mou Z.G., Hudak P.* An algebraic model for divide-and-conquer and its parallelism // Journal of Supercomputing. 1988. N 2. 257–278.
29. *Osoba F.O., Rabhi F.A.* A parallel multigrid skeleton using BSP // Proc. of EuroPar'98. LNCS. Berlin: Springer-Verlag. 1998. 1–6.
30. *Pelagatti S.* Structured Development of Parallel Programs. London–Bristol: Taylor & Francis, 1997.
31. *Pelagatti S.* A methodology for the development and the support of massively parallel programs: PhD thesis. Dipartimento di Informatica Universita' di Pisa. Pisa, 1993.
32. *Reynders J.V.W. et al.* POOMA: A framework for scientific simulations on parallel Architectures // In: Parallel Programming using C++. Cambridge: MIT Press, 1996. 1–43.
33. *Serot J., Ginhac D., Derutin J-P.* Skipper: A skeleton-based parallel programming environment for real-time image processing applications // LNCS. 1999. **1662**, 296–305.
34. *Siu S.* Openness and Extensibility in Design-Pattern-Based Parallel Programming Systems. Master of Applied Science Thesis. University of Waterloo. Ontario. Canada. Waterloo, 1996.
35. *Siu S., De Simone M., Goswami D., Singh A.* Design patterns for parallel programming // Parallel and Distributed Processing Techniques and Applications. California. Pasadena, 1996. 230–240.
36. *Skillicorn D.B., Hill J., McColl B.* Questions and answers about BSP // Scientific Programming. 1997. **6**, N 3. 249–274.
37. *Skillicorn D.B.* Models for practical parallel computation // International Journal of Parallel Programming. 1991. **20**, N 2. 133–158.
38. *Skillicorn D.B.* The Bird–Meertens formalism as a parallel programming model // NATO ASI Workshop on Software for Parallel Computation. Italy. Cetraro. June 1992. Berlin: Springer–Verlag, 1993. 1–14.
39. *Skillicorn D.B., Danelutto M., Pelagatti S., Zavanella A.* Optimizing data-parallel programs using the BSP cost model // LNCS. 1998. **1470**. 698–706.
40. *Snir M., Otto S., Huss-Lederman S., Walker D., Dongarra J.* MPI: The Complete Reference. Vol. 1. The MPI Core. Boston: MIT Press, 1998.
41. *Singh A., Schaeffer J., Szafron D.* Views on template-based parallel programming // Proc. of CASCON'96. Toronto, 1996. 1–12.
42. *To H.W.* Optimizing the Parallel Behavior of Combinations of Program Components: PhD thesis. University of London Imperial College of Science, Technology and Medicine, Department of Computing. London, 1995.
43. *Yabe T., Ishikawa T., Wang P.Y.* A universal solver for hyperbolic equations by cubic-polynomial interpolation II. Two- and three-dimensional solvers // Computer Physics Communications. 1991. N 66. 233–242.
44. *Zavanella A.* Optimizing skeletal stream parallelism on a BSP computer // LNCS. 1999. **1685**. 853-857.
45. *Zavanella A., Pelagatti S.* Using BSP to optimize data-distribution in skeleton programs // LNCS. 1999. **1593**. 613-622.
46. *Valiant L.G.* A bridging model for parallel computation // Communications of the ACM. 1990. **33**, N 8. 103–111.

Поступила в редакцию
17.01.2001