УДК 004.451.24

СТИРЕНКО С.Г.,
ЗИНЕНКО А.И.,
ГРИБЕНКО Д.В.

МОДЕЛЬ ОРГАНИЗАЦИИ ВЫЧИСЛЕНИЙ В РАСПРЕДЕЛЁННОЙ СИСТЕМЕ

Рассматриваются существующие модели организации вычислений в параллельных системах, применяемые в различных современных библиотеках и инструментальных средствах. Предложена усовершенствованная модель разбиения на независимые подзадачи, учитывающая размещение данных в локальной памяти узлов вычислительной системы. Приведен математический формализм и семантика операций, программная реализация которых позволит соответствовать математической модели.

Different models used in modern libraries and frameworks to organize computations in parallel systems are studied and reviewed. An improved task-splitting model compatible with work-stealing approach, which allows consideration of data locality in distributed system, is proposed. The mathematical formalism of this model and semantics of operations on the model are described.

Введение

Бурное развитие средств аппаратных средств вычислительной техники в последнее время ориентировано на создание высокопроизводительных систем путём увеличения количества используемых вычислительных элементов. Наибольшее распространение для решения вычислительно-ёмких научных задач получили кластерные системы как подмножество более широкого класса распределённых систем или систем с локальной памятью. Такие системы позволяют достаточно лёгкое горизонтальное масштабирование с целью наращивания вычислительных ресурсов. Однако, программирование таких систем требует значительных усилий по адаптации математически заданных алгоритмов к существующим моделям программирования, что приводит к значительному количеству ошибок при программировании, неэффективному распараллеливанию и, как следствие, неэффективному использованию вычислительных ресурсов [1]. Кроме того, некоторые подходы к представлению задач для систем с локальной памятью предполагают оптимизацию программы для существенно ограниченного объёма аппаратных ресурсов, что может привести к снижению отношения коэффициента ускорения к количеству используемых процессорных ядер при горизонтальном масштабировании [2].

Современные вычислительные средства предлагают несколько различных моделей программирования параллельных систем, в зависимости от доступного аппаратного обеспечения, в частности традиционное параллельное программирование для систем с общей памятью с использованием потоков, программирование распределённых систем через интерфейс передачи сообщений MPI, программирование ускорителей на базе Nvidia CUDA [3] или Intel Many Integrated Core architecture [4], и другие, каждая из которых имеет свои особенности. С другой стороны, предлагается ряд усовершенствованных подходов для описания вычислительных задач с использованием более высокого уровня абстракции, в частности автоматизированное распараллеливание с использованием OpenMP [5,6] или подход деления задач в Intel Threading Building blocks [7]. Однако, эти подходы не учитывают особенности распределённых систем, в особенности необходимость организации пересылок данных и оптимизацию их выполнения.

Таким образом, необходимо предложить модель программного интерфейса, которая бы использовала высокоуровневые абстракции задач и подзадач, тем самым избавляя пользователя от необходимости использовать особенности той или иной вычислительной системы, и учитывала особенности выполнения задач на распределённых системах с возможностью масштабирования. Такая модель предусматривает также возможность адаптивного подхода к изменению количества используемых ресурсов в процессе решения задачи и имеет возможность гибкой адаптации к размерности системы.

Подход к организации вычислений

Текущая ситуация, сложившаяся в сфере высокопроизводительных вычислений, диктует некоторые условия технологической реализации разрабатываемой системы, а именно:

• Доминирующее положение библиотеки MPI, которая обеспечивает коммуникацию между процессами, работающими на различных узлах, широкая поддержка и развитие её реализаций вендорами HPC систем и интерконнектов предопределяют использование MPI в разрабатываемой системе.

• По аналогичным причинам система изначально разрабатывается для Unix систем; тем не менее, размер платформо-зависимого кода небольшой, что в первую очередь обусловлено использованием MPI.

• Система предполагает проектирование и реализацию некоторого количества API, доступных программисту; для этого выбран язык C++ как язык, широко использующийся в HPC и обладающий мощными возможностями по созданию обобщённого кода благодаря шаблонам [8].

• Система представляет собой динамически загружаемую библиотеку, что даёт возможность не прилагать её копию к каждой программе, в то же время не требуя чрезмерных вмешательств в операционную систему как в случае с модулями ядра.

Способ описания заданий, подлежащих параллельному выполнению, сходен с применяемым в библиотеке TBB. Пользователю необходимо:

• выбрать один из готовых классов промежутков, подходящий для задачи, или описать свой (тип Range);

• описать класс, хранящий результат вычислений (тип Value);

• реализовать вычислительный алгоритм (оператор compute);

• реализовать алгоритм разбиения промежутков (оператор split);

• реализовать алгоритм слияния частичных результатов (оператор merge);

• реализовать алгоритм пересылок Range и Value между узлами.

Для запуска задания в параллельном режиме следует передать экземпляры всех этих классов в качестве аргументов функции parallel_reduce().

Не смотря на кажущееся большое количество вспомогательных сущностей и алгоритмов, все их элементы пользователь так или иначе реализовывал бы, выполняя распараллеливание вручную. Особенностью разрабатываемой библиотеки является лишь требование формализации и разделения этих сущностей.

Описанная пользователем задача, механизм её разделения и объединения результатов с помощью механизма шаблонов C++ превращается в код, использующий библиотеки MPI и Pthreads. Такой подход выгоден простотой использования и естественной поддержкой механизмов ООП, что является критичным при больших разработках [9].

Для выполнения пользовательской задачи система использует любой существующий MPI интеркоммуникатор на N процессов, каждый из которых содержит 2 потока: рабочий (worker thread, выполняет вычисления) и вор (thief thread, выполняет пересылки).

Каждый процесс имеет очередь промежутков, готовых для вычисления, а также очередь вычисленных частичных результатов. Когда очередь заданий пуста, рабочий поток пытается объединить какие-либо частичные результаты, иначе он будит поток-вор и блокируется, ожидая момента, когда поток-вор наполнит очередь заданий и разбудит его (подробности алгоритма, по которому наполняется очередь заданий, могут варьироваться в различных алгоритмах, соответствующих данной модели).

Семантика операций

Нижеописанная модель предполагает решение одной пользовательской задачи, описанной с использованием соответствующего программного интерфейса. Этот интерфейс реализует на некотором языке программирование функции, семантически эквивалентные нижеследующим абстрактным математическим операциям.

Под задачей T_0 будем понимать множество последовательностей действий $T_0 = \{t_{0,i}\}$, которые необходимо выполнить над некоторым набором входных данных. Каждая последовательность действий может состоять из произвольного числа действий $t_i = \langle a_0, a_1, a_2, \ldots \rangle$. Действия одной последовательности выполняются последовательно, т. е. так, как они бы выполнялись на однопроцессорной системе. Различные последовательности действий могут выполняться независимо и параллельно.

Исходная задача T_0 может быть разбита на подзадачи $\{T_1^{\langle 0 \rangle}, T_1^{\langle 1 \rangle}, T_1^{\langle 2 \rangle}, \ldots\}$ так, что множества последовательностей действий подзадач не пересекаются, а их объединение даёт изначальное

множество последовательностей действий. Пусть $T_1^{\langle j\rangle} = \{t_{1,i}^{\langle j\rangle}\}$, тогда

$$\bigcap_j T_1^{\langle j\rangle} = T_1^{\langle 0\rangle} \cap T_1^{\langle 1\rangle} \cap T_1^{\langle 2\rangle} \cap \ldots = \varnothing; \quad (1)$$

$$\bigcup_j T_1^{\langle j\rangle} = T_1^{\langle 0\rangle} \cup T_1^{\langle 1\rangle} \cup T_1^{\langle 2\rangle} \cup \ldots = T_0. \quad (2)$$

Более строгой версией (1), которая необходимо соблюдать в процессе разбиения, является

$$\{t_{1,a}^{\langle j\rangle}\} \cap \{t_{1,b}^{\langle j\rangle}\} \quad \forall a,b,j : a \neq b. \quad (3)$$

Подзадачи $T_1^{\langle j\rangle}$ называются *подзадачами первого порядка* или подзадачами исходной задачи. Для общности исходную задачу можем считать *подзадачей нулевого порядка*. Введём дополнительное ограничение на подзадачи: любая подзадача должна содержать хотя бы одну последовательность действий.

$$T_1^{\langle j\rangle} = \{t_{1,i}^{\langle j\rangle}\} \neq \varnothing \quad \forall j. \quad (4)$$

С другой стороны, если подзадача первого порядка содержит более одной последовательности действий, она также может быть разбита на *подзадачи второго порядка* с теми же условиями на множествах последовательностей действий

$$\begin{aligned} &T_1^{\langle j\rangle} = \bigcup_k T_2^{\langle j,k\rangle}, \\ &\bigcap_k T_2^{\langle j,k\rangle} = T_2^{\langle j,0\rangle} \cap T_2^{\langle j,1\rangle} \cap T_2^{\langle j,2\rangle} \cap \ldots = \varnothing. \end{aligned} \quad (5)$$

Вследствие (1), (2), из данных уравнений можем получить обобщённые условия для подзадач первого и второго порядка.

$$\bigcap_{j,k} T_2^{\langle j,k\rangle} = t_2^{\langle 0,0\rangle} \cap t_2^{\langle 0,1\rangle} \cap \ldots \cap t_2^{\langle 1,0\rangle} \cap t_2^{\langle 1,1\rangle} \cap \ldots = \varnothing. \quad (6)$$

Разбиение подзадач может быть продолжено рекурсивно до получения подзадач z-го порядка. Критерием остановки рекурсии является такое разбиение, которое в силу требования (4) может привести только к получению подзадачи $(z+1)$-го порядка из подзадачи z-го порядка, такой что

$$T_{(z+1)}^{\langle j,k,\ldots,l,0\rangle} = T_z^{\langle j,k,\ldots,l\rangle}. \quad (7)$$

Такую подзадачу назовём *неделимой*. Для неделимых задач дальнейшее разбиение не имеет смысла.

Операторы над подзадачами

Введём обобщённый оператор разбиения **gsplit** над подзадачами: данный оператор выполняет разбиение подзадачи z-го порядка на множество подзадач $(z+1)$-го порядка

$$\text{gsplit } T_z^{\langle j,\ldots,k\rangle} \to \{T_{(z+1)}^{\langle j,\ldots,k,i\rangle}\}, \quad i \in \overline{(1, N_{(z+1)}^{\langle j,\ldots,k\rangle})}, \quad (8)$$

где $N_{(z+1)}^{\langle j,\ldots,k\rangle}$ - количество подзадач, на которые была разбита исходная подзадача $T_{(z+1)}^{\langle j,\ldots,k\rangle}$. Оператор может выполнять разбиение любыми доступными способам при соблюдении следующих условий:

$$\begin{cases} T_{(z+1)}^{\langle j,\ldots,k,a\rangle} \cap T_{(z+1)}^{\langle j,\ldots,k,b\rangle} = \varnothing \quad \forall a \neq b; \\ \bigcup_i T_{(z+1)}^{\langle j,\ldots,k,i\rangle} = T_z^{\langle j,\ldots,k\rangle}; \\ N_{(z+1)}^{\langle j,\ldots,k\rangle} > 1; \end{cases} \quad (9)$$

которые являются обобщениями (1), (2) и (4) для произвольной подзадачи.

Концепцию обобщённого оператора разбиения и подзадач можно преобразовать следующим образом для лучшего соответствия практическим применениям. Часто на практике необходимо выполнить одинаковую последовательность действий $\langle a_0, a_1, a_2, \ldots\rangle$ над некоторым множеством данных $\{r_i\}$. Чаще всего набор является упорядоченным и последовательным, т. е. имеет смысл говорить о *промежутке* решения $[r_l, r_h)$. Промежуток эквивалентен конечному счётному множеству данных $\{r_i\}$.

Таким образом, задача может быть представлена в виде пары из последовательности действий и промежутка, над которым необходимо выполнить действие

$$T_0 = (\langle t_i\rangle, [r_l, r_h)). \quad (10)$$

Такая запись предполагает выполнение одинаковой последовательности действий над всеми элементами промежутка или его частями. Эквивалентность данной записи изначальной концепции задачи может быть достигнута путём использования условных операторов внутри последовательности действий.

В виду достаточно большой сложности последовательности действий и её неизменности целесообразно обобщить её и представить в виде *оператора вычисления* **compute**, выполняющего определённые вычисления над произвольной частью промежутка с целью получения некоторого значения:

$$\text{compute}[r_a, r_b) \to v_{[r_a, r_b)}. \quad (11)$$

Покажем эквивалентность двух форм задания подзадачи. Пусть изначально имелась зада-

ча T_0^a, представленная конечным множеством последовательностей действий $T_0^a = \{t_i\}, i \in \overline{(1,N)}$. Определим промежуток $R = [1, N+1), R \in \mathrm{N}$. И зададим оператор вычисления как

$$\mathrm{compute}\,[a,b) = \bigcup_{i=a}^{b} t_i.$$

Альтернативная форма записи задачи примет вид $T_0^b = (\mathrm{compute}, R)$. Оба вида определения задач после вычисления приводят к одинаковому множеству результатов, таким образом, возможно эквивалентное преобразование между данными двумя формами записи задач.

Определим оператор разбиения **split**, как частный случай обобщённого оператора **gsplit**, для промежутков.

$$\mathrm{gsplit}(\mathrm{compute}, R) \to \{(\mathrm{compute}, R_i)\},$$

где части промежутка $R_i = \mathrm{split}\, R$ определяется при помощи оператора разбиения

$$\mathrm{split}\; R_z^{\langle j,\dots,k\rangle} \to \{R_{(z+1)}^{\langle j,\dots,k,i\rangle}\}, \quad i \in \overline{(1, N_z^{\langle j,\dots,k\rangle})}. \quad (12)$$

Представим каждый промежуток в виде $R_z^{\langle j,\dots,k\rangle} = \left[r_{z,l}^{\langle j,\dots,k\rangle}, r_{z,h}^{\langle j,\dots,k\rangle}\right)$ Тогда условия (9) примут вид

$$\begin{cases} r_{(z+1),l}^{\langle j,\dots,k,0\rangle} = r_{z,l}^{\langle j,\dots,k\rangle} \\ r_{(z+1),h}^{\langle j,\dots,k,N_z^{\langle j,\dots,k\rangle}\rangle} = r_{z,h}^{\langle j,\dots,k\rangle} \\ r_{z,l}^{\langle j,\dots,k,i\rangle} = r_{z,h}^{\langle j,\dots,k,(i-1)\rangle} \quad \forall i \in \overline{(2, N_z^{\langle j,\dots,k\rangle})} \\ r_{z,h}^{\langle j,\dots,k,i\rangle} > r_{z,l}^{\langle j,\dots,k,i\rangle} \forall i \end{cases} \quad (13)$$

Понятие пути разбиения и слияния

Последовательное применение оператора разбиения к подзадачам, представленным как в виде множеств последовательностей действий, так и в виде промежутков, приводит к формированию у каждой подзадачи некоторой последовательности индексов Г: $T_z^{\langle \gamma_1, \gamma_2, \gamma_3, \dots\rangle}$, где длина последовательности определяет порядок подзадачи, а каждый элемент обозначает номер подзадачи данного порядка, из которой путём разбиения была получена данная. *Путь разбиения* – это последовательность индексов подзадач i-го порядка $\Gamma = \langle \gamma_i \rangle$, из которых была получена данная подзадача. Для исходной задачи путь разбиения является пустой последовательностью $\langle\rangle$. Путь разбиения позволяет уникальным образом идентифицировать задачу, определить источник исходных данных, и назначение результирующих данных.

Применение понятия пути разбиения позволяет выявить дополнительные свойства разбиений на основе (9), в частности:

$$T_z^{\langle \gamma_1, \dots \gamma_n, \gamma_{n+1} \dots \gamma_N \rangle} \subset T_y^{\langle \gamma_1, \dots \gamma_n \rangle} \quad \forall n,$$

причём $z = y + N - n$. Аналогично

$$T_z^{\langle \gamma_1, \dots \gamma_n, \gamma_a, \dots \gamma_N \rangle} \cap T_y^{\langle \gamma_1, \dots \gamma_n, \gamma_b \dots \gamma_N \rangle} = \varnothing$$

$\forall n, a, b : a \neq b.$

Задачу после разбиения можно представить в виде дерева подзадач, где корнем является исходная задача, а узлами и листьями – подзадачи. Тогда для каждого узла можно идентифицировать последовательность подзадач, которым необходим результат вычисления данной подзадачи путём отсечения последних элементов пути разбиения.

Введём оператор слияния **merge**. Данный оператор выполняет объединение результатов вычислений подзадач одного порядка.

$$\mathrm{merge}\,\{V_{(z+1)}^{\langle j,\dots,k,i\rangle}\} \to V_z^{\langle j,\dots,k\rangle} \quad i \in \overline{(1, N_z^{\langle j,\dots,k\rangle})} \quad (14)$$

Объединение возможно только одновременно всех результатов подзадач, порождённых из данной подзадачи. Таким образом, оператор слияния может быть некоммутативен.

Однако, поскольку задача может быть представлена в виде произвольного дерева подзадач и результаты могут быть готовы в произвольные моменты времени, необходимым условием корректной работы оператора слияния является его ассоциативность.

Задание, которое может быть выполнено разрабатываемой системой параллельно, может быть представлено в виде кортежа:

$$(R, \mathrm{compute}, \mathrm{split}, \mathrm{merge}), \quad (15)$$

где операторы вычисления, разбиения и слияния действуют над полями

$$\mathrm{split} \quad = R \to \{R\};$$

$$\mathrm{compute} \quad = R \to V;$$

$$\mathrm{merge} \quad = \{V\} \to V;$$

и R, V – типы промежутка входных данных и результата соответственно.

Способы предварительного разбиения на подзадачи

Рассмотренная модель предполагает использование техники динамической реорганизации графа подзадач: изначально задача или её некоторая часть разбивается на подзадачи, каждая

из которых может быть дополнительно разделена на меньшие подзадачи в процессе выполнения. Широко известно, что граф задачи является её неотъемлемым свойством, а алгоритмы планирования стремятся оптимизировать соответствие этого графа графу вычислительной системы, что приводит к возможности двух базовых решений. Соответственно, граф задачи известен статически. В отличие от такого подхода, данная модель предполагает обобщённый механизм описания возможности разбиения задачи на части, который, тем не менее, не применяется для полной декомпозиции задачи. Модель описывает общий шаблон графа подзадач, а его реальное инстранцирование происходит в процессе выполнения, что позволяет использовать адаптивные подходы. Выполнение одной и той же задачи на одной и той же системе может приводить к различным инстанцированиям графа подзадач, в зависимости от решений, принятых в процессе работы системы.

Для данных целей мы используем две основных реализации оператора разбиения:

• равномерное разбиение – подзадача z-го порядка разбивается на меньшие подзадачи $(z+k)$-го порядка (рис. 1);

• адаптивное разбиение – подзадача z-го порядка разбивается подзадачи $(z+1)$-го порядка, часть их которых разбивается на подзадачи $(z+2)$-го порядка и т. д. до получения неделимых задач (рис. 2).

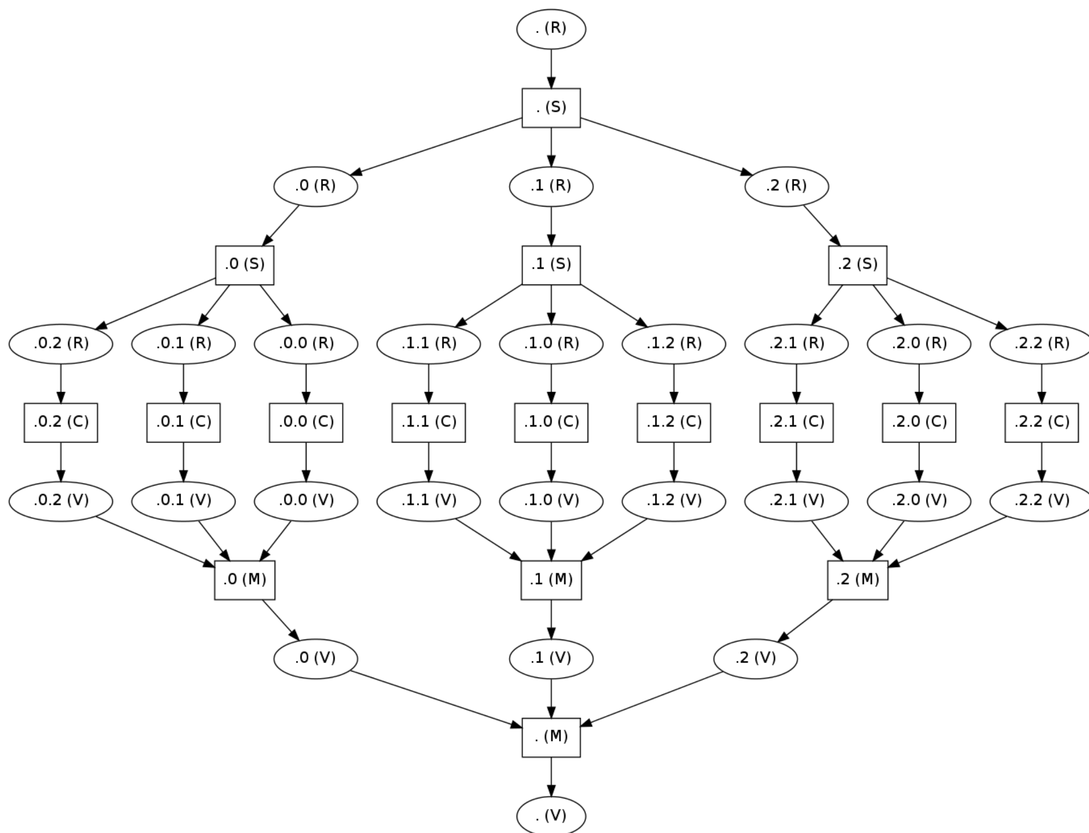

Рисунок 1. Пример разбиения задачи на подзадачи третьего порядка.

Равномерное разбиение

Равномерное разбиение характеризуется числом увеличения порядка подзадач k. Альтернативно его можно выразить через количество требуемых подзадач $k = \log_2 N$.

В данной работе исследуются два возможных варианта количества подзадач:

• разбиение на <<крупные>> подзадачи (PreSplitLargest), т. е. на минимальное количество подзадач N, такое что $N \geq P$, где P – число процессоров. Таким образом каждый процессор в идеальном случае получит ровно одну подзадачу, которая может необходимо выполнить. В таком случае ожидаются высокие коэффициенты ускорения и эффективности за счёт осуществления минимально возможного количества пересылок данных между процессорами. Тем не менее, в случае невозможности разбиения на строго необходимое количество частей ($N = P$) возможна сильная неравномерность нагрузки на процессоры, что может значительно снизить ускорение и эффективность параллельного выполнения.

- разбиение на «средние» подзадачи (PreSplitMid), т. е. на количество подзадач $N \geq P^2$. Этот способ имеет два преимущества: во-первых, часто меньшая по количеству вычислений подзадача требует меньшего количества данных, что позволит начать вычисления как можно раньше; во-вторых, нагрузка на процессоры будет лучше сбалансирована, так как неравномерность нагрузки может составлять не более времени вычисления одной задачи плюс время пересылки исходных данных. Выбор квадратичной зависимости определяется возможностью алгоритмов планирования по запросам забирать задачи из очереди другого процессора. Действительно, если каждый процессор изначально имеет в своей очереди P задач, то одну он может взять на выполнение, а оставшиеся по одной могут забрать остальные процессы остальными процессами.

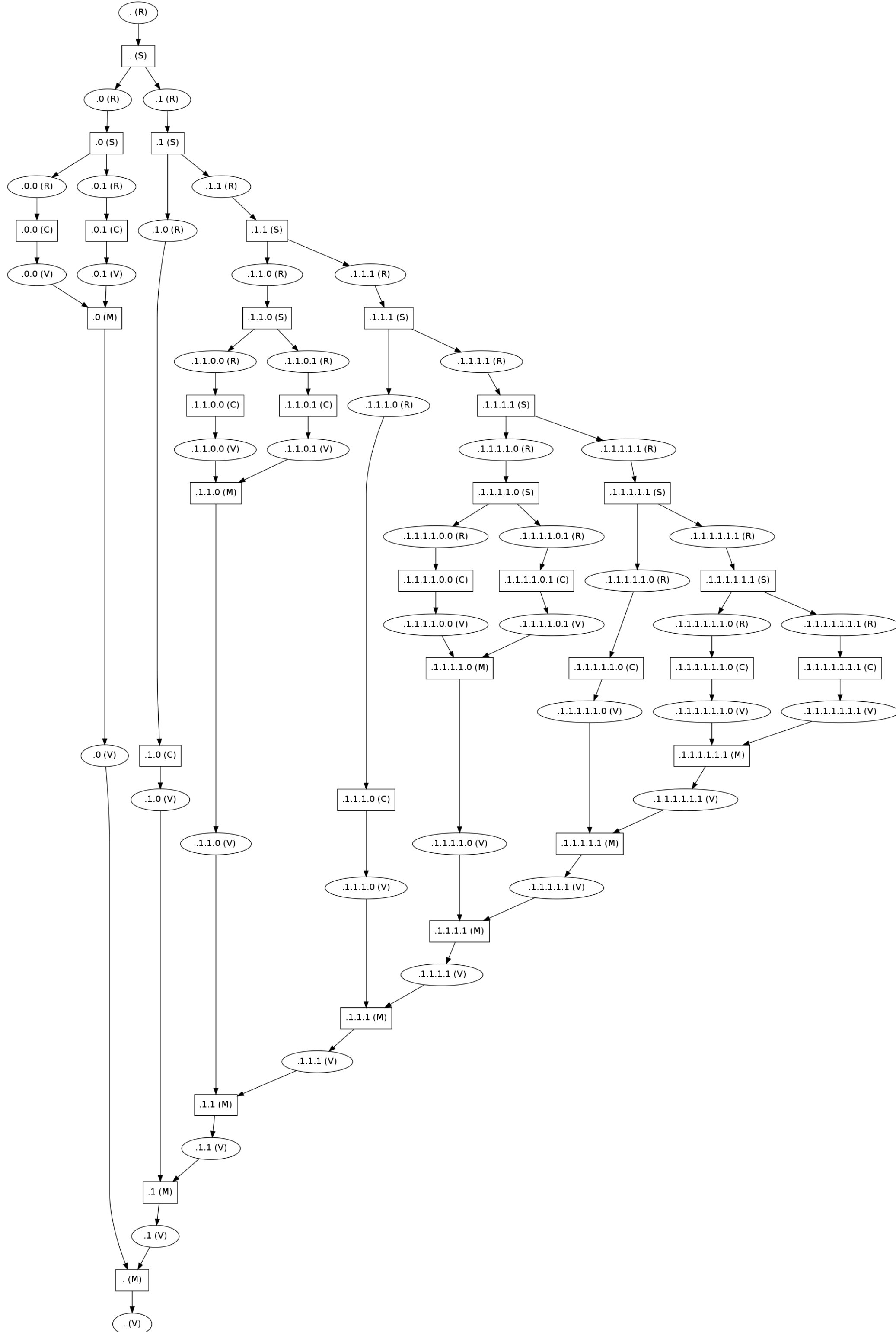

Рисунок 2. Пример динамического адаптивного разбиения задачи.

Адаптивное разбиение

Адаптивный подход к разбиению (PreSplitAdaptive) разработан с целью решения проблемы неравномерности ожидаемой нагрузки на процессоры при предварительном разбиении. Изначально он разбивает исходную задачу на минимально возможное количество подзадач, большее или равное количеству процессоров. После этого на каждый процессор назначается по одной такой подзадаче, а оставшиеся повторно разбиваются по аналогичному принципу. Разбиение заканчивается если не осталось неназначенных подзадач, т. е. каждый процессор получил в сумме одинаковые наборы подзадач, или оставшиеся задачи являются неделимыми. В худшем случае неравномерность нагрузки составит одну неделимую задачу. Данный алгоритм предварительного разбиения может быть достаточно эффективен в случае задач с равномерной вычислительной сложностью на промежутке, однако может привести к посредственным результатам в случае случайного распределения вычислительной сложности по промежутку (впрочем, как и любой другой алгоритм предварительного разбиения).

Варианты операторов разделения и слияния

Оператор разбиения **split** может быть определён для некоторой задачи $T_0 = (\text{compute}, [r_l, r_h])$ на промежутке любым способом, который обеспечивает соблюдение условий (9). Аналогичным образом определяется и оператор слияния **merge**.

В случае представления задачи в виде промежутка исходных данных целесообразно все подзадачи также представлять в виде промежутков для сохранения однородности операторов. Возможно и различное представление подзадач других порядков, однако это существенно усложняет операторы разделения и слияния из-за введения условных ветвлений. Промежуток, как правило, включает все допустимые значения в заданном интервале, что также должно соблюдаться и для промежутков подзадач большего порядка.

Таким образом, целесообразно выполнять разбиение промежутка на ряд последовательных частей. Поскольку алгоритмы строятся в предположении, что вычисления неделимых подзадач приблизительно равны по вычислительной сложности оптимальным представляется разбиение промежутка на части равных размеров.

Наиболее простым подходом является деление промежутка на две части (рис. 3), таким образом всего может существовать до 2^i подзадач i-го порядка (рис. 1). Если представить иерархию подзадач в виде дерева (рис. 2), то видно, что не всегда каждая ветвь данного дерева имеет одинаковую глубину, т. е. не всегда задачи разбиваются до одинакового порядка подзадач. Следовательно, в некоторых случаях возможно получение $N = \sum_i a_i 2^i$ задач, где $a_i \in \mathrm{N}$ – некоторые целочисленные коэффициенты. Можно показать, что это уравнение относительно неизвестных a_i имеет неотрицательное решение для любых положительных N.

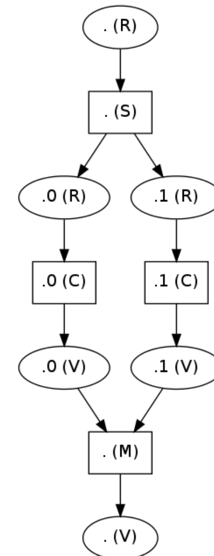

Рисунок 3. Схема разбиения на две подзадачи

Рассмотрим также разделение на три одинаковые части (рис. 4). Аналогично может существовать не более 3^i подзадач i-го порядка. Также возможно образование $N = \sum_i a_i 3^i$ задач. Рассмотрим детально это уравнение. Пусть изначально существует одна подзадача нулевого порядка. Её можно разделить на три части, т. е. заменить одну подзадачу тремя, увеличив таким образом количество подзадач на 2. Аналогичным образом любая подзадача z-го порядка может быть заменена на 3 подзадачи $(z+1)$-го порядка, увеличивая суммарное количество подзадач на 2. Таким образом, N может принимать только нечётные значения, что может сказаться на балансировании нагрузки на процессоры.

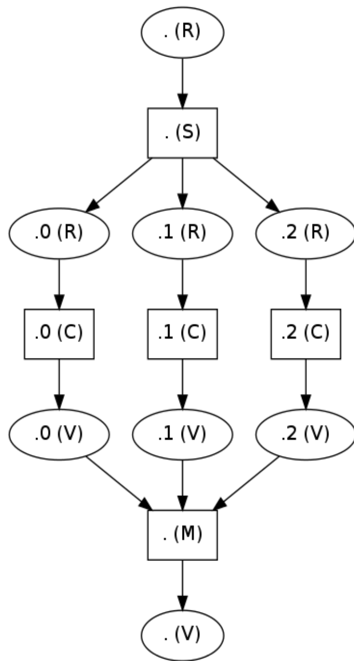

Рисунок 4. Схема разбиения на три подзадачи

Разделение на четыре одинаковые части может быть получено двойным применением разделения на две. Однако, используя рассуждения, аналогичные предложенным для разбиения на три части, можно установить, что суммарное количество подзадач в данном случае может составлять $1,4,7,10,13,\ldots$.

Обобщая рассмотрим разделение на k одинаковых частей. k конечно, т. к. промежуток конечного размера может быть разделен только на конечное число неделимых частей (в случае существования условия неделимости). Если перед выполнением разделения очередной подзадачи существовало N_a подзадач, то после замены одной из них на k новых суммарное количество подзадач станет равным $N_b = N_a + k - 1$. Таким образом, любое количество разделений данного типа может привести только к $N = N_i + j(k-1)$ подзадачам, где $N_i = 1$ – начальное количество задач, $j \in \mathrm{N}$ – некоторая константа.

Для разделения на k количество подзадач должно удовлетворять условию $N \equiv 1 \bmod (k-1)$. Для $k = 2$ может быть получено любое количество задач, а с ростом k множество возможных значений уменьшается. Следовательно, рассматривать разделение более чем на 3 части нецелесообразно.

Выводы

В последнее время широкое распространение получили модели программирования распределённых систем с использованием высокоуровневых абстракций, например подзадач. Такой подход позволяет с одной стороны скрыть от пользователя сложность внутренней организации аппаратной компоненты современных распределённых вычислительных систем, т. к. пользователь получает возможность описывать задачу в терминах некоторых абстрактных математических операций, а с другой – получить возможность использовать одно и то же описание задачи с использованием некоторого программного интерфейса для работы на вычислительных системах со значительно различающимися архитектурными организациями.

Существующие модели представления задач при помощи подобных абстракций не предполагают использование систем с локальной памятью в виду того, что не учитывают пересылки и локальность данных. Предложенная модель расширяет одну из существующих моделей таким образом, чтобы исправить этот недостаток.

Предложенная модель использует обобщённые математические операции для описания пользовательской задачи и возможности её представления для параллельного вычисления. Данная модель абстрагирует особенности организации конкретной вычислительной системы и позволяет описывать лишь задачу и корректный способ передачи данных для этой задачи. Динамическое конструирование графа подзадач позволит в будущем использовать адаптивные подходы при выборе способа разбиения задачи на подзадачи и использовать эту информацию для улучшения адаптивных алгоритмов динамического планирования.

Предложенная система представляет собой математическую модель, которая может быть использована для анализа и изучения процесса выполнения параллельных программ в распределённых системах кластерного типа. В дальнейшем предполагается расширить данную модель таким образом, чтобы более полно учесть пересылки данных при выполнении задачи, а также учесть вероятностные свойства динамического выполнения подзадач и передачи данных между ними.

Список литературы:

1. *Bastoul Cedric.* Contributions to High-Level Program Optimization // Habilitation (Dr.Sc.) Thesis. Paris-Sud University, France. – 2012.
2. Automatic parallelization and locality optimization of beam forming algorithms [Text] / Albert Hartono, Nicolas Vasilache, Cedric Bastoul *et al.* // High Performance Embedded Computing Workshop (HPEC). – MIT Lincoln Laboratory, Lexington, Massachusetts: 2010.
3. *Kirk D.* Nvidia CUDA software and gpu parallel computing architecture [Text]: Tech. rep.: Nvidia, 2007.
4. Evaluation and improvements of programming models for the Intel SCC many-core processor [Text] / C. Clauss, S. Lankes, P. Reble, T. Bemmerl // Proceedings of International conference on High Performance Computing and Simulation (HPCS), 2011. – 2011. – Pp. 525–532.
5. Board OpenMP Architecture Review. OpenMP Application Program Interface Version 3.1 [Text]. – 2011.
6. *Marowka Ami.* Performance of OpenMP benchmarks on multicore processors [Text] // Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing. – ICA3PP ’08. – Berlin, Heidelberg: Springer-Verlag, 2008. – Pp. 208–219.
7. Intel Threading building blocks documentation [Electronic resource]. – Access mode: http://threadingbuildingblocks.org/documentation.php. – Last access: 08.05.2012. – Title from the screen.
8. JTCL.22.32 – ISO/IEC 14882 international standard: Programming language c++ [Electronic resource]. – Access mode: http://www.open-std.org/jtc1/sc22/wg21/. – Last access: 01.03.2013. – Title from the screen.
9. MPI: A message-passing interface standard [Electronic resource]. – Access mode: http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf. – Last access: 08.05.2012. – Title from the screen.