Введение в параллельное программирование

- ***Параллельные вычисления*** – способ организации компьютерных вычислений, при котором программы разрабатываются, как набор взаимодействующих вычислительных процессов, работающих асинхронно и при этом одновременно.
- ***Параллельное программирование*** (ПП) – это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (*multithreading*).

Многоядерные вычисления

ПП позволяет максимально эффективно использовать возможности многоядерных процессоров и многопроцессорных систем.

По ряду причин (повышение потребления энергии, ограничения пропускной способности памяти) увеличивать тактовую частоту процессоров стало невозможно. Вместо этого стали увеличивать ***производительность*** процессоров за счет размещения в одном чипе нескольких вычислительных **ядер**, не меняя (или снижая) тактовую частоту.

Поэтому для увеличения скорости работы приложений следует по-новому подходить к организации кода, а именно – оптимизировать программы под многоядерные системы.

Множественные потоки команд/данных (Классификация М. Флинна)

Самой ранней и наиболее известной является классификация архитектур вычислительных систем, предложенная в 1966 году М. Флинном.

Она базируется на понятии **потока**, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. На основе числа потоков команд и потоков данных **Флинн** выделяет четыре класса архитектур: ***SISD, MISD, SIMD, MIMD*** (табл. 1.1)

Таблица 1.1. Описание классов архитектур

Название класса	Описание класса
SISD (single instruction stream, single data stream) или ОКОД (Одиночный поток Команд, Одиночный поток Данных)	Исполнение одним процессором одного потока команд, обрабатывающего данные, хранящиеся в одной памяти.
SIMD (single instruction stream, multiple data stream) или ОКМД (одиночный поток команд, множественный поток данных)	Один поток команд, включающий векторные команды, что позволяет выполнять одну арифметическую операцию сразу над многими данными – элементами вектора.
MISD (multiple instruction stream, single data stream) или МКОД (Множественный поток Команд, Одиночный поток Данных)	Подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных.
MIMD (multiple instruction stream, multiple data stream) или МКМД (Множественный поток Команд, Множественный поток Данных)	Предполагает, что в вычислительной системе есть несколько устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных.

На основании Табл. 1.1 можно проранжировать архитектуры на однопоточность/многопоточность (Табл. 1.2).

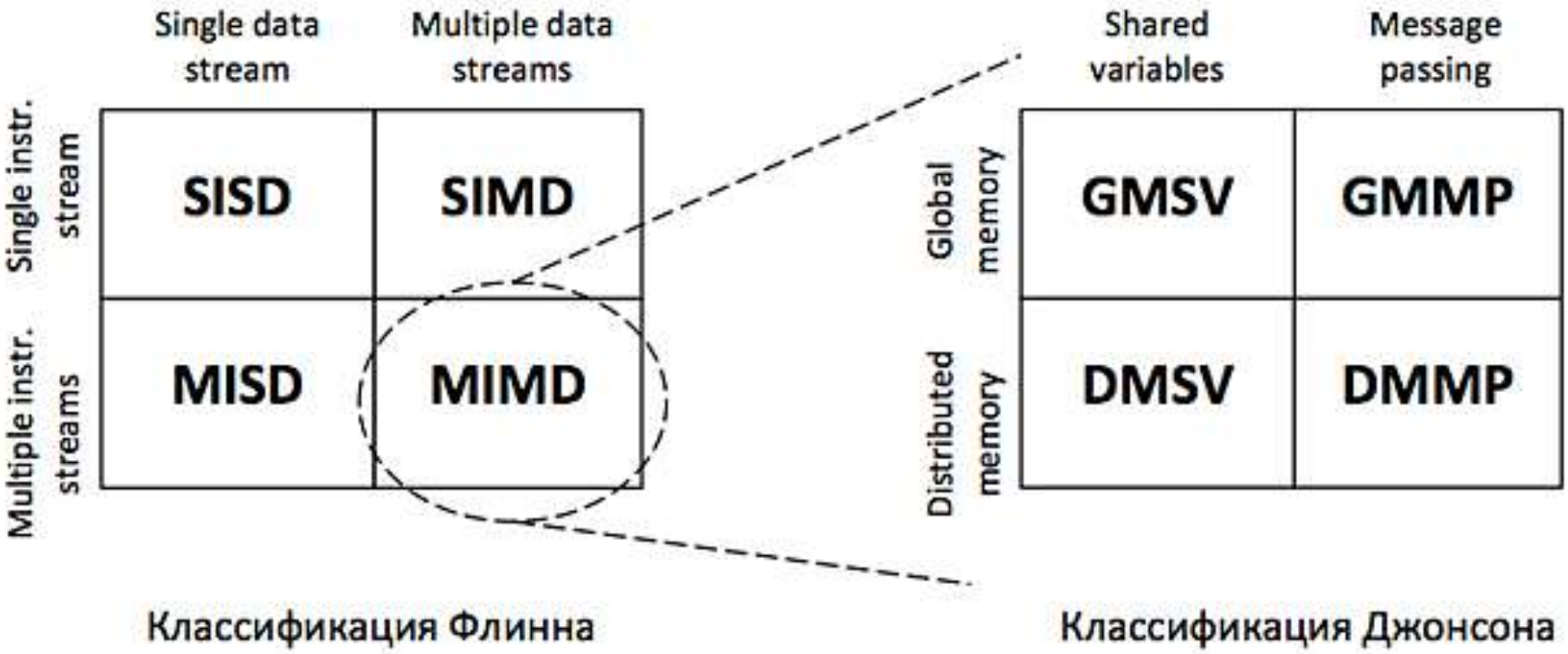
Таблица 1.2. Ранжирование архитектур по обработке потоков

	Одиночный поток команд (Single Instruction)	Множество потоков данных (Multiple Data)
Одиночный поток данных (Single Data)	SISD (ОКОД)	SIMD (ОКМД)
Множество потоков команд (Multiple Instruction)	MISD (МКОД)	MIMD (МКМД)

Классификация Флинна относит почти все параллельные вычислительные системы к одному классу – *MIMD*.

Для выделения разных типов параллельных вычислительных систем применяется **классификация Джонсона**, в которой дальнейшее разделение многопроцессорных систем основывается на используемых способах организации оперативной памяти в этих системах.

Данный подход позволяет различать два важных типа многопроцессорных систем: multiprocessors (мультипроцессорные или системы с общей разделяемой памятью) и multicomputers (мультикомпьютеры или системы с распределенной памятью).



Global memory – глобальная память

Distributed memory – распределенная память

Shared variables – разделяемые переменные

Message passing – передача сообщений

Классификация Джонсона основана на структуре памяти (глобальная или распределенная) и механизме коммуникаций и синхронизации (разделяемые переменные или передача сообщений).

Системы GMSV (global-memory-shared-variables) часто называются также мультипроцессорами с разделяемой памятью (shared-memory multiprocessors).

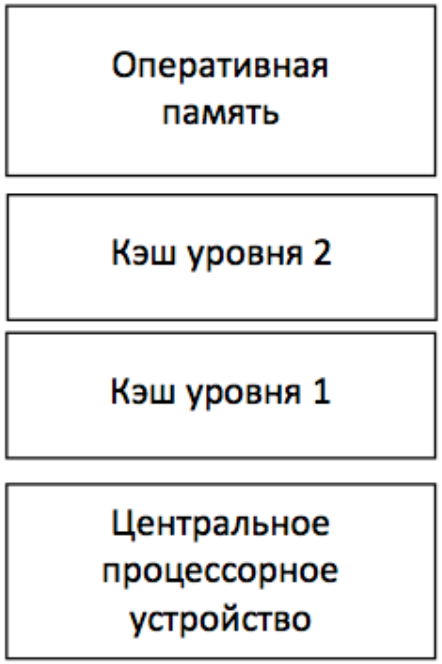
Системы DMMP (distributed-memory-message-passing) также называемые мультикомпьютерами с распределенной памятью (distributed-memory multicomputers).

Архитектура однопроцессорной машины

Современная однопроцессорная машина:

- центральное процессорное устройство (ЦПУ)
- первичная память
- один или несколько уровней *кэш*-памяти (*кэш*)
- вторичная (дисковая) память
- набор периферийных устройств (дисплей, мышь, клавиатура, модем, CD, принтер и т.д.)

Основными компонентами для выполнения программ являются ЦПУ, кэш и память.



Мультикомпьютеры с распределенной памятью

В мультикомпьютерах с распределенной памятью существуют соединительная **сеть**, но каждый *процессор* имеет собственную **память**. Соединительная *сеть* поддерживает передачу сообщений.

Мультикомпьютеры (многопроцессорные системы с распределенной памятью) не обеспечивают общий доступ ко всей имеющейся в системах памяти. Каждый процессор системы может использовать только *свою локальную память*, для доступа к данным, располагаемых на других процессорах, необходимо использовать интерфейсы передачи сообщений (например, стандарт *MPI*). Данный подход используется при построении многопроцессорных вычислительных систем – массивно-параллельных систем и кластеров.

В мультикомпьютере процессоры и сеть расположены физически близко (в одном помещении). Он одновременно используется одним или небольшим числом приложений; каждое приложение задействует выделенный набор процессоров. Соединительная сеть с большой пропускной способностью предоставляет высокоскоростной путь связи между процессорами.

Мультипроцессор с разделяемой памятью

В мультипроцессоре и в многоядерной системе исполнительные устройства (процессоры и ядра процессоров) имеют доступ к разделяемой оперативной памяти. Процессоры совместно используют оперативную память. У каждого процессора есть собственный *кэш*.

Если процессоры ссылаются на разные области памяти, их содержимое можно безопасно поместить в *кэш* каждого. Если процессоры обращаются к одной области памяти, то в случае когда оба процессора только считывают данные, в *кэш* каждого помещается копия данных. Когда один из процессоров записывает в память, возникает *проблема согласованности кэш*а: в кэш-памяти другого процессора теперь содержатся неверные данные.

Обеспечение однозначности кэшей реализуется на аппаратном уровне – для этого после изменения значения общей переменной все ее копии в кэшах отмечаются как недействительные и последующий доступ к переменной потребует обязательного обращения к основной памяти.

Режимы выполнения независимых частей программы

При рассмотрении проблемы организации параллельных вычислений различают следующие возможные режимы выполнения независимых частей программы:

1. Режим разделения времени (многозадачный режим).
2. Распределенные вычисления.
3. Синхронные параллельные вычисления.

1. Режим разделения времени (многозадачный режим)

Предполагает, что число подзадач (процессов или потоков одного процесса) больше, чем число исполнительных устройств. Активным (исполняемым) может быть одна единственная подзадача, а все остальные процессы (потоки) находятся в состоянии ожидания своей очереди на использование процессора; использование режима разделения времени может повысить эффективность организации вычислений, кроме того в данном режиме проявляются многие эффекты параллельных вычислений (необходимость взаимоисключения и синхронизации процессов и др.).

Многопоточность приложений в операционных системах с разделением времени применяется даже в однопроцессорных системах. Она повышает отзывчивость приложения – если *основной поток* занят выполнением каких-то расчетов или запросов, другой *поток* позволяет реагировать на действия пользователя. Каждый поток может планироваться и выполняться независимо.

Пример. Когда пользователь нажимает кнопку мышки, посылается сигнал процессу, управляющему окном, в котором в данный момент находится курсор мыши. Этот процесс (*поток*) может выполняться и отвечать на щелчок мыши. Приложения в других окнах могут продолжать при этом свое выполнение в фоновом режиме.

2. Распределенные вычисления

Компоненты выполняются на машинах, связанных локальной или глобальной сетью. Процессы взаимодействуют, обмениваясь сообщениями. Такие системы пишутся для:

- распределения обработки (файловые серверы),
- обеспечения доступа к удаленным данным (БД в Web),
- интеграции и управления данными, распределенными по своей сути (в промышленных системах),
- повышения надежности (отказоустойчивые системы).

Многие распределенные системы организованы как системы типа *клиент-сервер*. Например, *файловый сервер* предоставляет файлы данных для процессов, выполняемых на клиентских машинах.

3. Синхронные параллельные вычисления

Их цель – быстро решать данную задачу или за то же время решить большую задачу.

Пример

- научные вычисления, моделирующие такие явления, как глобальный климат, эволюция солнечной системы, результат действия нового лекарства;
- графика и обработка изображений, спецэффекты;
- крупные комбинаторные или оптимизационные задачи (планирование авиаперелетов, экономическое моделирование).

Программы решения таких задач требуют эффективного использования доступных вычислительных ресурсов. Число подзадач должно быть оптимизировано с учетом числа исполнительных устройств (процессоров, ядер).

Уровни параллелизма в многоядерных архитектурах

- *Параллелизм на уровне команд* позволяет процессору выполнять несколько команд за один такт (переупорядочивание команд оптимальным образом с целью исключения остановки вычислительного процесса и увеличения количества команд, выполняемых за один такт).
- *Параллелизм на уровне потоков процесса* – позволяет выделить независимые потоки исполнения команд в рамках одного процесса. Поддерживается на уровне ОС, которая распределяет потоки процессов *по ядрам процессора* с учетом приоритетов и максимально задействует свободные ресурсы.
- *Параллелизм на уровне приложений* – одновременное выполнение нескольких программ. Достигается за счет выделение каждому приложению кванта процессорного времени.

Анализ эффективности параллельных вычислений

- **Ускорение (Speedup) параллельного алгоритма**:

$$S_n = \frac{T_1}{T_n}$$

где T_n – время вычисления задачи на n процессорах, T_1 – время выполнения однопоточной программы.

T(n) < T(1), если параллельная версия алгоритма эффективна.

T(n) > T(1), если накладные расходы (издержки) реализации параллельной версии алгоритма чрезмерно велики.

- **Эффективность параллельного алгоритма:**

$$E_n = \frac{S_n}{n}$$

Идеальное теоретическое значение $E_1 = 1$ (алгоритм достигает максимального ускорения $S_n = n$).

На практике эффективность убывает при увеличении числа процессоров.

$E_n > 1$ чаще всего потому, что:

- в качестве последовательного алгоритма был применен не самый быстрый алгоритм из существующих.
- с увеличением количества вычислителей растет суммарный объем их оперативной и кэш памяти. Все большая часть данных умещается в оперативной памяти и не требует подкачки с диска, или умещается в кэше.

Закон Амдала

Закон Амдала (1967 г) описывает теоретический выигрыш в производительности параллельного решения по отношению к лучшему последовательному решению:

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

где S_n – во сколько раз можно ускорить вычисления (ускорение), n – количество процессоров (ядер), α – доля последовательно вычисляемого кода ($\alpha \neq 0$).

Закон Амдала накладывает ограничения на максимально достижимую эффективность параллельного алгоритма.

Предположим, например, что α=1/3, то есть две трети операций в алгоритме могут выполняться параллельно, а треть – нет. Тогда ускорение $S_n < 3$. Таким образом, независимо от количества процессоров (ядер) и даже при игнорировании всех затрат на подготовку данных нельзя ускорить решение задачи более, чем в три раза.

Закон Густафсона-Барсиса

Закон Густафсона-Барсиса (1988 г.) оценивает максимально допустимое ускорение выполнения параллельной программы, в зависимости от количества одновременно выполняемых потоков вычислений и доли последовательных расчётов.

Формула Густафсона-Барсиса:

$$S_n = n + (1 - n)\alpha,$$

где α – доля последовательных расчётов в программе, n – количество процессоров.

Густафсон заметил, что пользователи склонны к изменению тактики решения задачи, работая на многопроцессорных системах.

Например, на 100 процессорах программа выполняется 20 минут, на 1000 процессорах можно достичь времени исполнения порядка 2 минут.

Однако для получения большей точности решения имеет смысл увеличить **объём** решаемой задачи, т.е. при сохранении общего времени исполнения пользователи стремятся получить более точный результат. Увеличение объёма решаемой задачи приводит к увеличению доли параллельной части, так как последовательная часть (ввод/вывод, менеджмент потоков, точки синхронизации и т.п.) не изменяется.

Проблемы разработки параллельных приложений

Декомпозиция – разбиение задачи на относительно независимые части (подзадачи).

Декомпозиция задачи может быть проведена несколькими способами:

1. По заданиям.
2. По данным.
3. По информационным потокам.

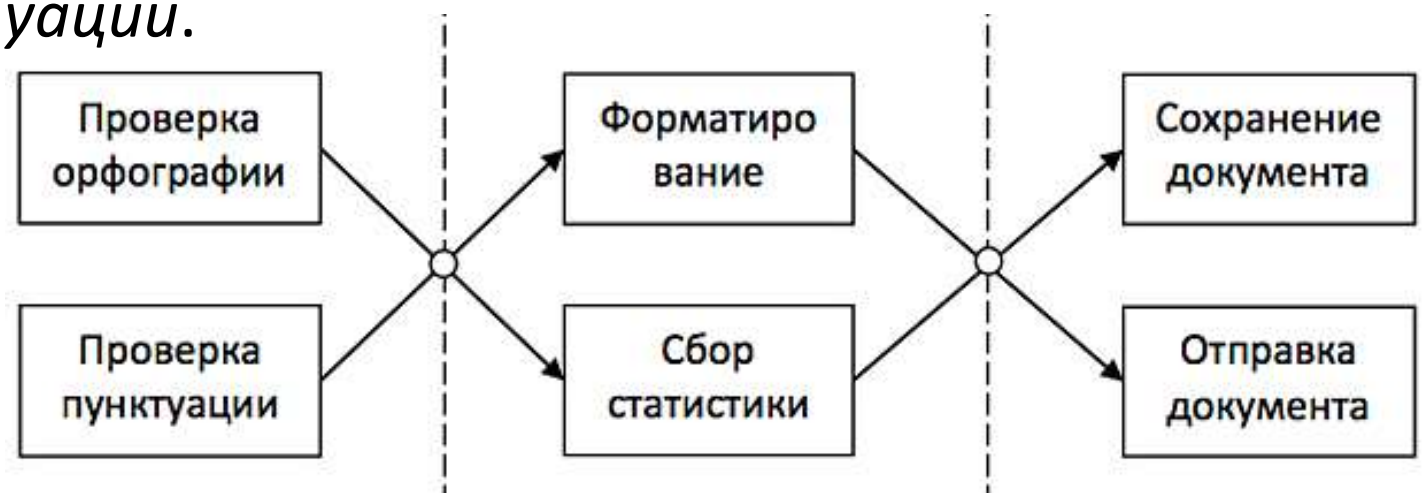
1. Декомпозиция по заданиям (функциональная декомпозиция)

предполагает присвоение разным потокам разных функций.

Например, приложение выполняет правку документа и включает следующие функции:

- проверка орфографии (CheckSpelling),
- проверка пунктуации (CheckPuncto),
- форматирование текста (Format),
- подсчет статистики по документу (CalcStat),
- сохранение изменений в файле (SaveChanges),
- отправка документа по электронной почте (SendDoc).

Функциональная декомпозиция разбивает работу приложения на подзадачи таким образом, чтобы каждая подзадача была связана с отдельной функцией. Но не все операции могут выполняться параллельно. Например, *сохранение* документа и *отправка* документа выполняются только после завершения всех предыдущих этапов. *Форматирование* и *сбор статистики* могут выполняться параллельно, но только после завершения *проверки орфографии и пунктуации*.



2. Декомпозиции по данным

При *декомпозиции по данным* каждая подзадача работает со своим фрагментом данных.

В рассматриваемом примере *декомпозиция по данным* может применяться к задачам, допускающим работу с фрагментом документа.

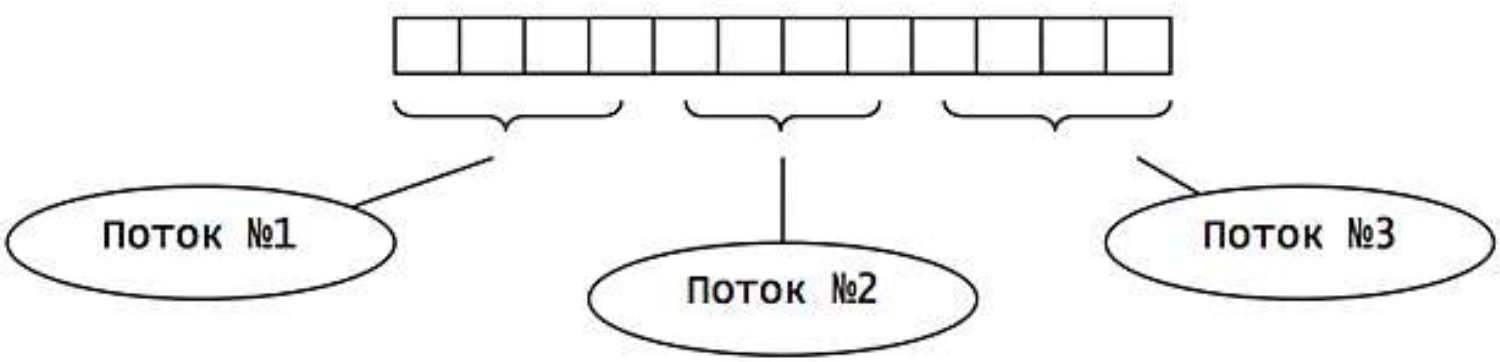
Функции CheckSpelling, CheckPuncto, CalcStat, Format объединяются в одну подзадачу, но создается несколько экземпляров этой подзадачи, которые параллельно работают с разными фрагментами документа. Функции SaveChanges и SendDoc составляют отдельные подзадачи, так как не могут работать с частью документа.

Принципы разделения данных между подзадачами:

- статический,
- динамический.

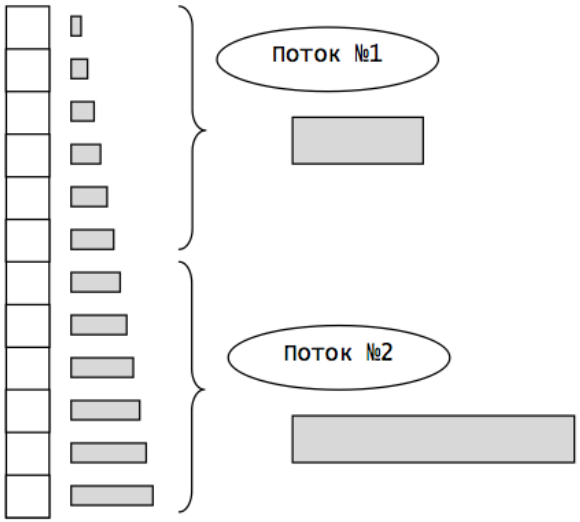
При *статической декомпозиции* фрагменты данных назначаются потокам до начала обработки и, как правило, содержат одинаковое число элементов для каждого потока.

Например, разделение массива элементов может осуществляться по равным диапазонам индекса между потоками (range partition).



Основное достоинство статического разделения – независимость работы потоков. Эффективность статической декомпозиции снижается при разной вычислительной сложности обработки элементов данных.

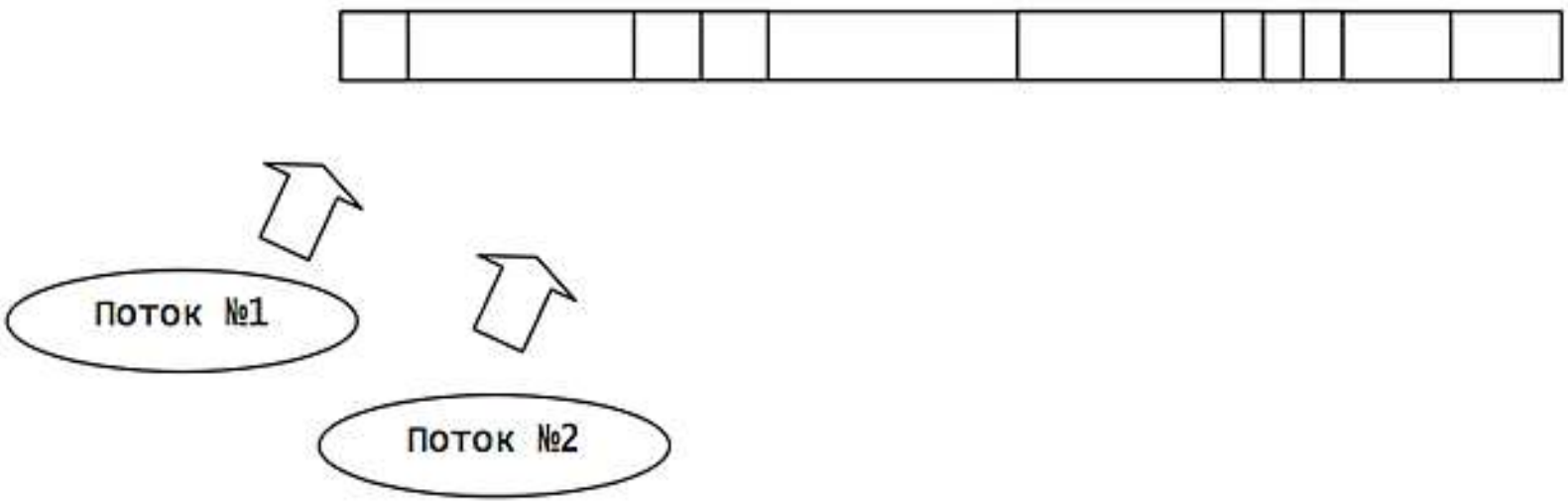
Например, вычислительная нагрузка обработки *i*-го элемента массива может зависеть от его индекса.



Несбалансированность нагрузки разных потоков снижает эффективность распараллеливания.

Решение: *первый поток* обрабатывает все четные элементы, *второй поток* обрабатывает все нечетные.

Когда вычислительная сложность обработки элементов заранее не известна, сбалансированность загрузки потоков обеспечивает *динамическая декомпозиция*.



При *динамической декомпозиции* каждый поток обращается за блоком данных (порцией). После обработки блока данных поток обращается за следующей порцией. Динамическая декомпозиция требует *синхронизации* доступа потоков к структуре данных. Размер блока определяет частоту обращений потоков к структуре. Некоторые алгоритмы динамической декомпозиции увеличивают размер блока в процессе обработки. Если поток быстро обрабатывает элементы, то размер блока для него увеличивается.

3. Декомпозиция по информационным потокам

выделяет подзадачи, работающие с одним типом данных.

В рассматриваемом примере могут быть выделены следующие подзадачи:

- работа с черновым документом (орфография и пунктуация):
- работа с исправленным документом (форматирование и сбор статистики);
- работа с готовым документом (сохранение и отправка).

Масштабирование подзадач

Свойство *масштабируемости* – эффективное использование всех имеющихся вычислительных ресурсов, тесно связано с выбранным алгоритмом решения задачи. Один алгоритм очень хорошо распараллеливается, но только на *две* подзадачи, другой алгоритм позволяет выделить *произвольное* число подзадач.

Обязательное условие масштабируемости приложения – возможность *параметризации* алгоритма в зависимости от числа процессоров в системе и от текущей загруженности системы. Она позволяет изменять число выделяемых подзадач при конкретных условиях выполнения алгоритма.

Модели параллельных приложений

- модель делегирования ("управляющий-рабочий");
- сеть с равноправными узлами;
- конвейер;
- модель "производитель-потребитель"

Каждая модель характеризуется собственной декомпозицией работ, которая определяет, кто отвечает за порождение подзадач и при каких условиях они создаются, какие информационные зависимости между подзадачами существуют.

Модель	Описание
Модель делегирования	Центральный поток ("управляющий") создает "рабочие" потоки и назначает каждому из них задачу, ожидает завершения работы потоков и собирает результаты.
Модель с равноправными узлами	Все потоки имеют одинаковый рабочий статус.
Конвейер	Применяется для поэтапной обработки потока входных данных.
Модель "производитель-потребитель"	Частный случай конвейера с одним производителем и одним потребителем.