Теория вычислительных процессов

Process calculus

Параллельное программирование

Черновик

ИВТ и ПМ
ЗабГУ

2021

Outline

Зачем?

- Если задача не решается за достаточное время на одном компьютере (ядре, процессоре, ...) то их нужно увеличить число компьютеров
- Особенно много времени могут занимать научные и инженерные вычисления
  - Моделирование физических процессов, прочностные расчёты
  - Фолдинг белков, SETI, климатические вычисления
  - БАК в секунду генерирует 100Тб данных
  - BigData и DataScience

Top500 supercomputers (ноябрь 2021)

Зачем?

- Серверы и другие сетевые программы тоже требует параллельного выполнения задач
- Банковские процессы (обработка транзакций)
- Компьютерная графика
- Машинное обучение
- ...

вычисления в кино

- Lord of the Rings (2001-2003)
  - company: Weta Digitals
  - 3200 processor cluster
  - single scene contains
    - per second 24 frames
    - per frame: 4996 x 3112 points with 32- or 64 bit color encoding
    - each object means a separate compute cycle

Зачем?

- Фоновое выполнение задач в программах с интерфейсом пользователя

  Фоновые вычисления тем более эффективны, потому что практически все современные устройства (ПК, мобильные устройства) имеют многоядерные процессоры, не говоря уже о серверах и суперкомпьютерах.

План

1. Введение в || программирование
2. Потоки в C++.
3. Взаимодействие, синхронизация и обмен данными.
4. OpenMP (Open Multi-Processing). Параллельные вычисления на C++
5. MPI (Message Passing Interface)
6. Вычисления на GPU (python - pyTorch, google colab)
7. ???
8. Доклады по желанию

Не будет рассмотрена организация процессов и потоков в ОС

Ссылки

- CSC курс || программирования: wiki.osll.ru/doku.php/courses:high_performance_computing : *start*
- Параллельное программирование на C++ в действии. Практика разработки многопоточных программ, 2016, Энтони Уильямс

- Ваши любимые книги?

Асинхронный и синхронный процессы

Асинхронный процесс – процесс, в котором операция не требует отклика от другой операции для своего выполнения.

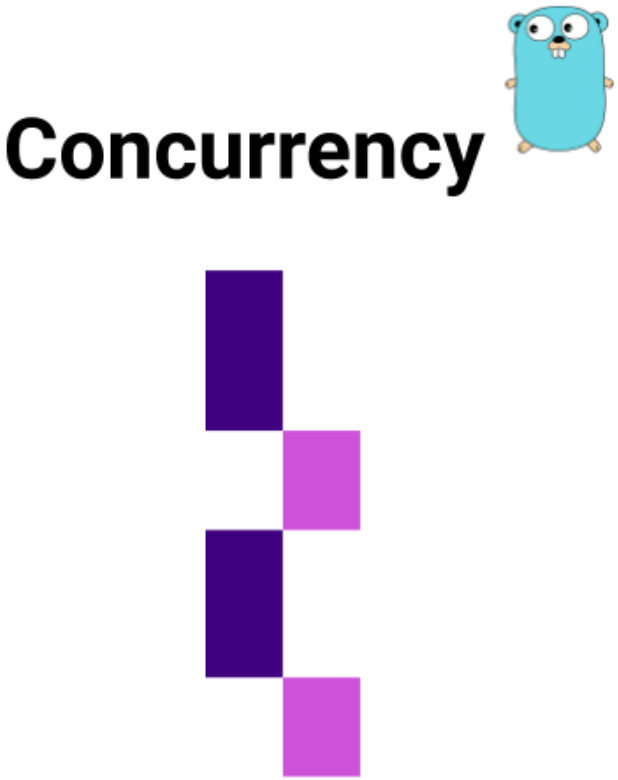
Синхронный процесс – процесс, в котором операция требует отклика для продолжения выполнения.

Concurrency vs Parallelism

Concurrency is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. For example, multitasking on a single-core machine.

Parallelism is when tasks literally run at the same time, e.g., on a multicore processor. is when tasks literally run at the same time, e.g., on a multicore processor.

Concurrency vs Parallelism



Concurrency vs Parallelism

Пример concurrency: попеременная (причем порядок заранее не определён) работа нескольких потоков на одном ядре процессора.

Пример параллелизма: одновременная работа потоков на нескольких ядрах процессора.

Concurrency тоже может дать прирост в производительности, например за счёт того, что ожидание ресурсов может быть выделено в отдельный поток.

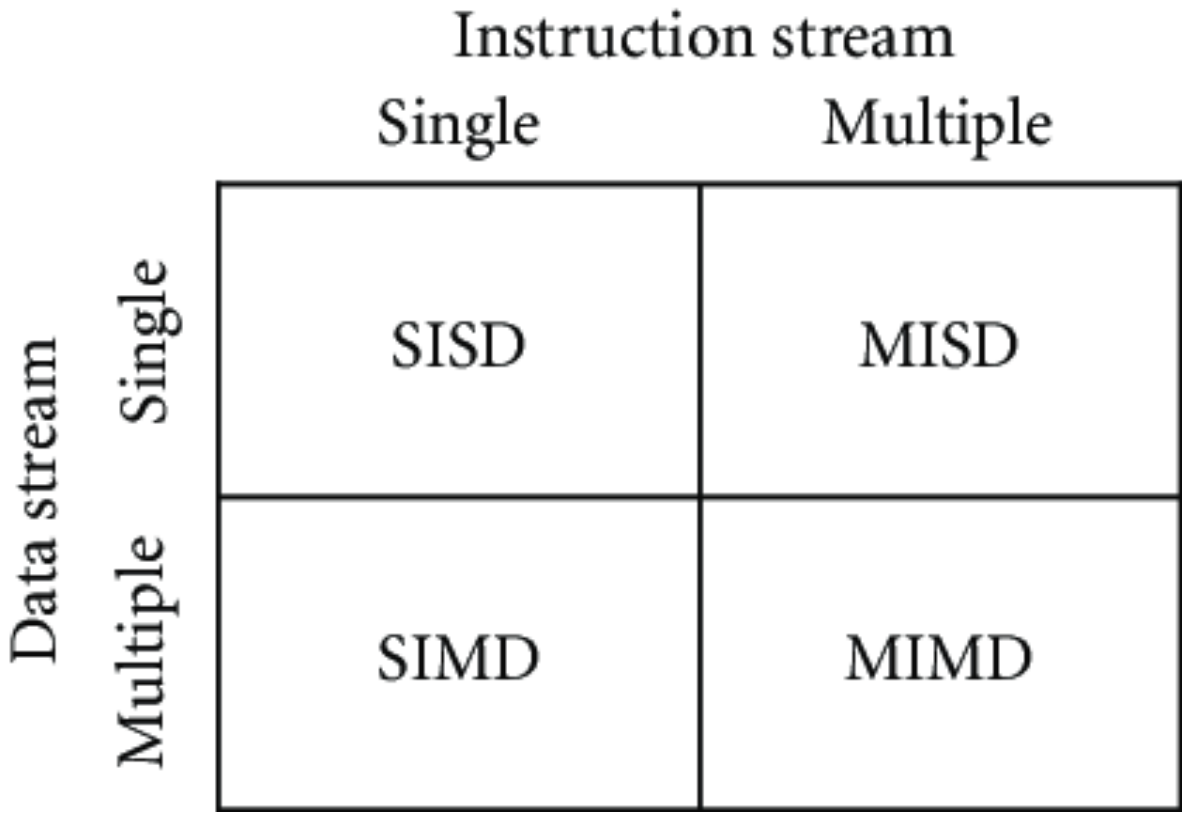
Режимы выполнения программ

- Многозадачный режим (режим разделения времени)
  Параллельности нет. Программы выполняются попеременно и последовательно. Выигрыш в быстродействии за счёт отсутствия блокирующих программу ожиданий.
- Параллельный режим
- Распределённое выполнение то же самое что и параллельный режим, только выполнение на физически раздельных устройствах с существенными временными задержками на синхронизацию.

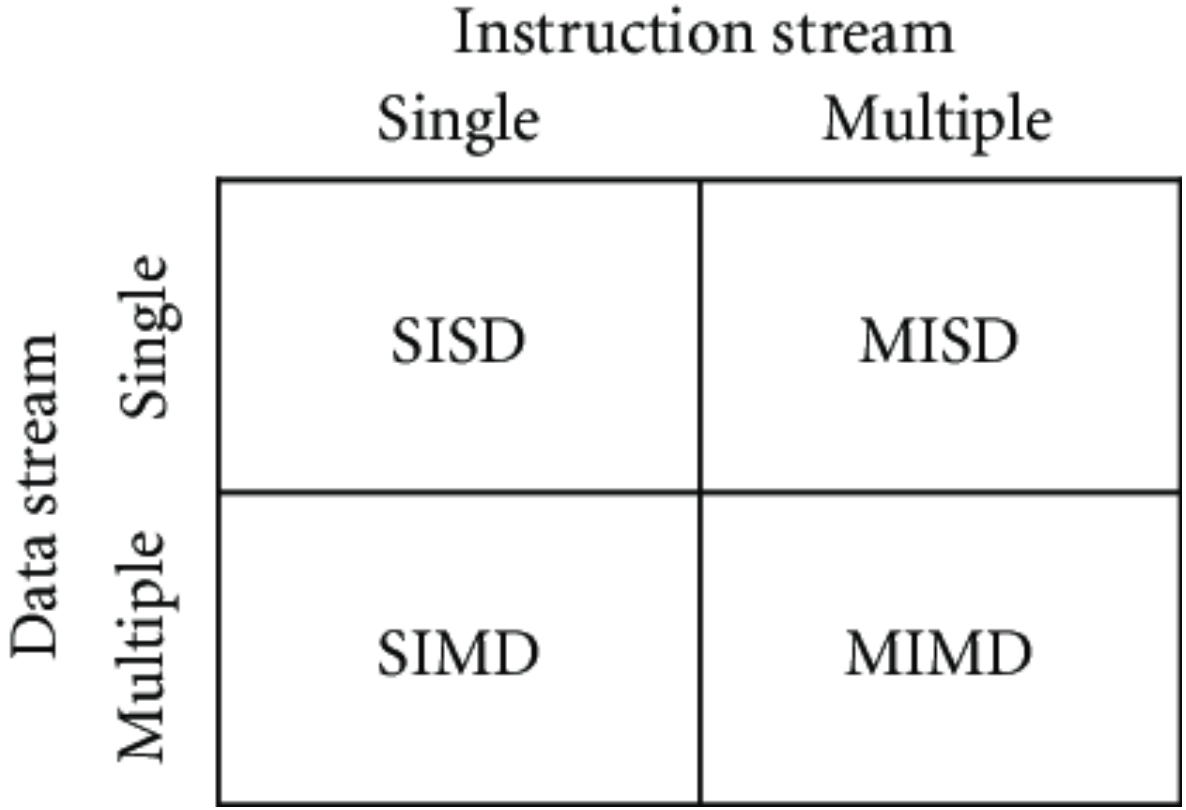
Outline

Классификация Флинна (Flynn)

- классификация архитектур ЭВМ по признакам наличия параллелизма в потоках команд и данных
- Предложена предложена Майклом Флинном в 1960-х



Классификация Флинна (Flynn)



- SISD, single instruction stream over a single data stream
- SIMD, single instruction, multiple data
- MISD, multiple instruction, single data
- MIMD, multiple instruction, multiple data

SISD

- Классическая архитектура фон Неймана.
- || отсутствует
- Все однопроцессорные (с одним ядром) системы
- Частный случай SISD – конвейер

Вычислительный конвейер (Instruction pipelining)

Instr. No. \ Clock cycle	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

- Конвейер позволяет достичь равномерной скорости выполнения всего цикла операций.
- Как будет отличатся частота выполнения цикла операций (из 5 этапов) на конвейере и с помощью 5 параллельных обработчиков?

MISD

- Используется для резервирования
- Применяется в системах критичных к сбоям (самолёты, космические аппараты, системы SCADA[1])

[1] Supervisory Control And Data Acquisition — диспетчерское управление и сбор данных

SIMD

- Одновременное выполнение одной инструкции над разными данными
- Векторный процессор — это процессор, в котором операндами некоторых команд могут выступать упорядоченные массивы данных — векторы
- В большинстве современных микропроцессоров имеются векторные расширения (SSE и др)
- Большинство современных графических процессоров

SSE

- SSE (Streaming SIMD Extensions, потоковое SIMD-расширение процессора) — это SIMD набор инструкций, разработанный Intel и впервые представленный в процессорах серии Pentium III как ответ на аналогичный набор инструкций 3DNow! от AMD.
- SSE – только один из примеров векторных расширений

SSE

перемножение четырёх пар чисел с плавающей точкой одной инструкцией mulps

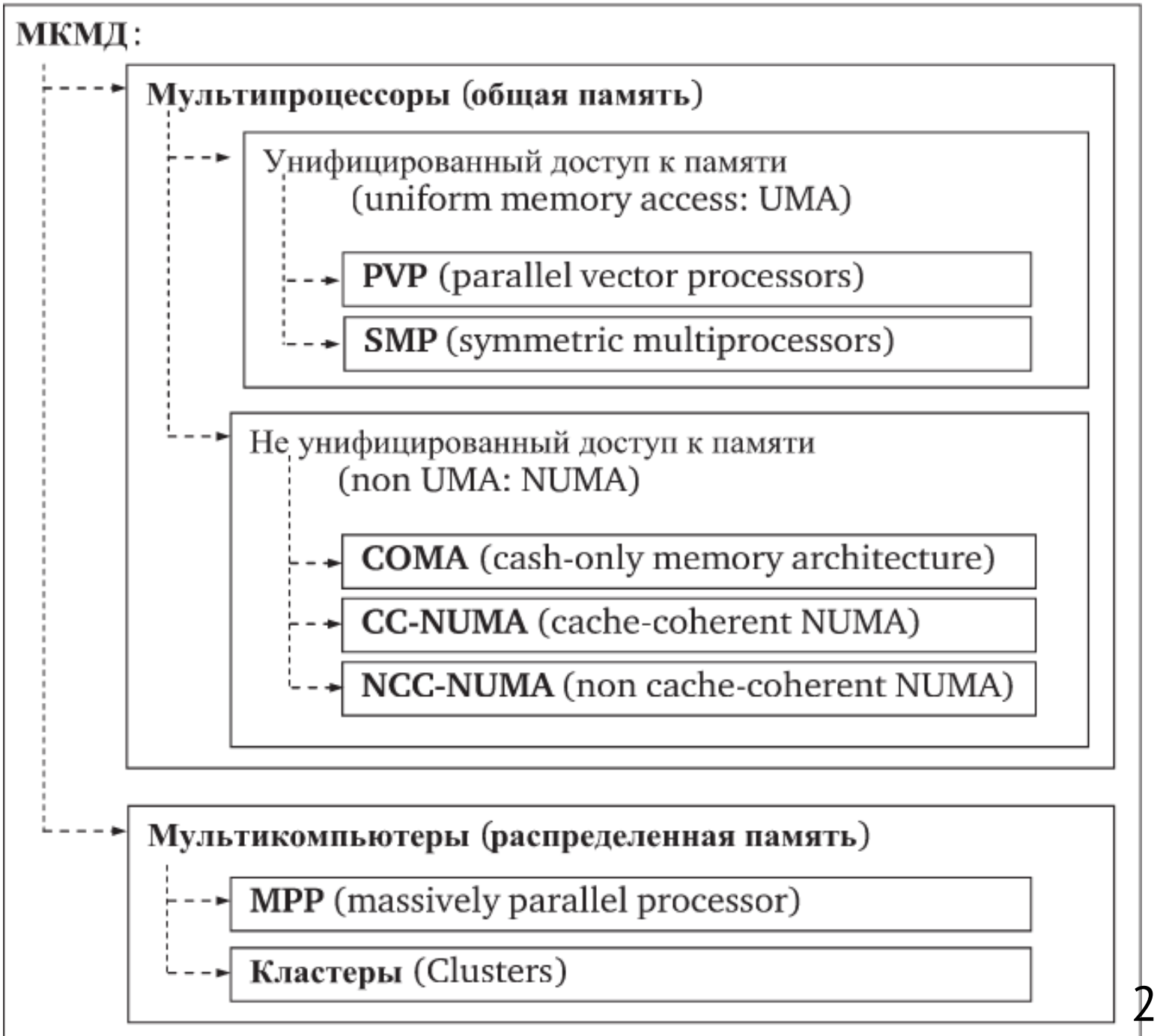
```
__declspec(align(16)) float a[4] = { 300.0, 4.0, 4.0, 12.0 };
__declspec(align(16)) float b[4] = {   1.5, 2.5, 3.5,  4.5 };
__asm {
    movups xmm0, a    ; //поместить 4 перем. из a в рег-р xmm0
    movups xmm1, b    ; //поместить 4 перем. из b в рег-р xmm1
    mulps xmm0, xmm1  ; //перемножить пакеты перем:
                        //xmm0 = xmm0 * xmm1
                      ; // xmm00 = xmm10 * xmm00
                      ; // xmm01 = xmm11 * xmm01
                      ; // xmm02 = xmm12 * xmm02
                      ; // xmm03 = xmm13 * xmm03

    movups a, xmm0    ; // выгрузить результаты
                        // из регистра xmm0 по адресам a
};
```

MIMD

- Разные вычислительные устройства выполняют свои программы на своими данными
- Почти все суперкомпьютеры.

MIMD


МКМД:
Мультипроцессоры (общая память)
Унифицированный доступ к памяти
(uniform memory access: UMA)
PVP (parallel vector processors)
SMP (symmetric multiprocessors)
Не унифицированный доступ к памяти
(non UMA: NUMA)
COMA (cash-only memory architecture)
CC-NUMA (cache-coherent NUMA)
NCC-NUMA (non cache-coherent NUMA)
Мультикомпьютеры (распределенная память)
MPP (massively parallel processor)
Кластеры (Clusters)

Классификация 2

- Системы с общей памятью
  - Легко программировать
  - Не нужно пересылать сообщения
  - Трудно масштабировать
  - Конкуренция за доступ к общим ресурсам

  - Многопоточность, OpenMP
- Системы с распределённой памятью
  - Отказоустойчивы
  - Сложнее программировать
  - Нужен обмен данными
  - Легко масштабировать

  - Несколько процессов, MPI

Outline

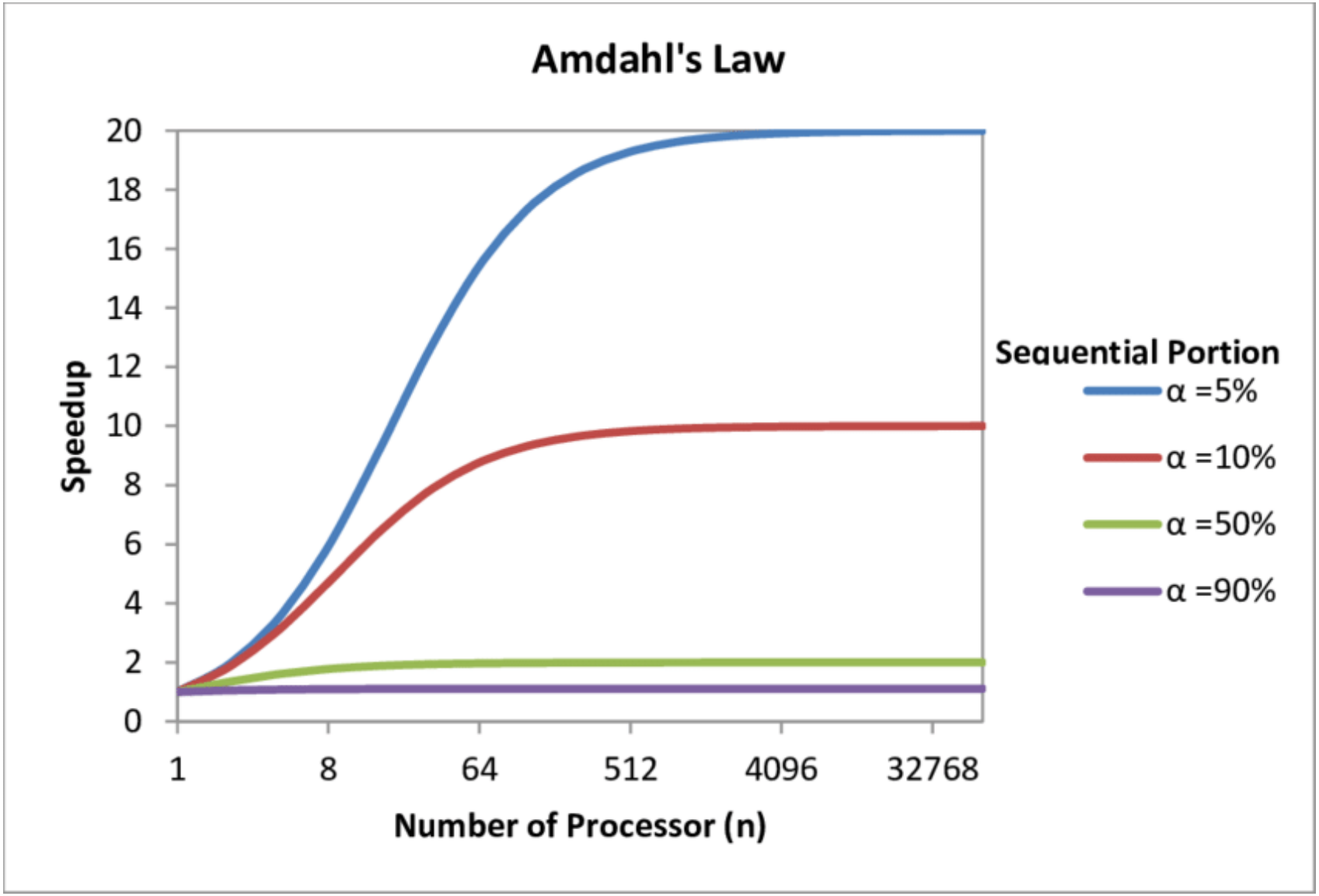
Закон Амдала

ускорение, которое может быть получено на вычислительной системе из процессоров, по сравнению с однопроцессорным решением не будет превышать величины

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

- α – доля последовательных вычислений
- p – число процессов

Закон Амдала



Закон Амдала

- Закон Амдала даёт только верхнюю оценку ускорению, фактическое же ускорение будет ниже из-за накладных расходов на создание процессов и синхронизации.
- Плохо поддаются распараллеливанию рекурсивные и итерационные алгоритмы

С чего стоит начинать распараллеливание программы?

- При распараллеливании алгоритма можно в первом приближении пользоваться правилами оптимизации.
- Правило 90/10: 90% времени выполнения программы занимает выполнение всего 10% кода.
- Значительную часть этих 10% занимают циклы

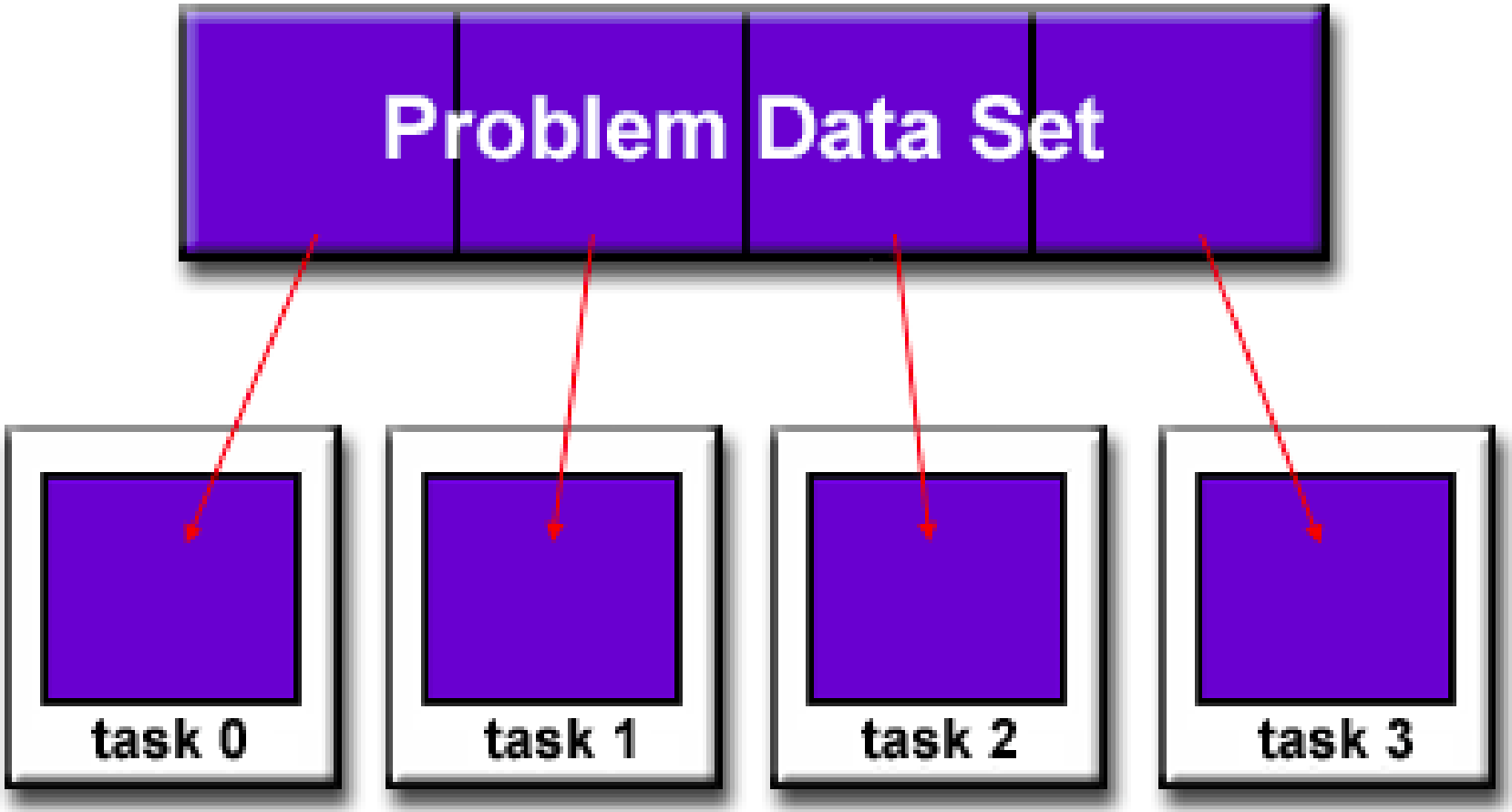
См. также оптимизацию циклом компилятором

Outline

Декомпозиция

- По данным
- По инструкциям

Декомпозиция по данным



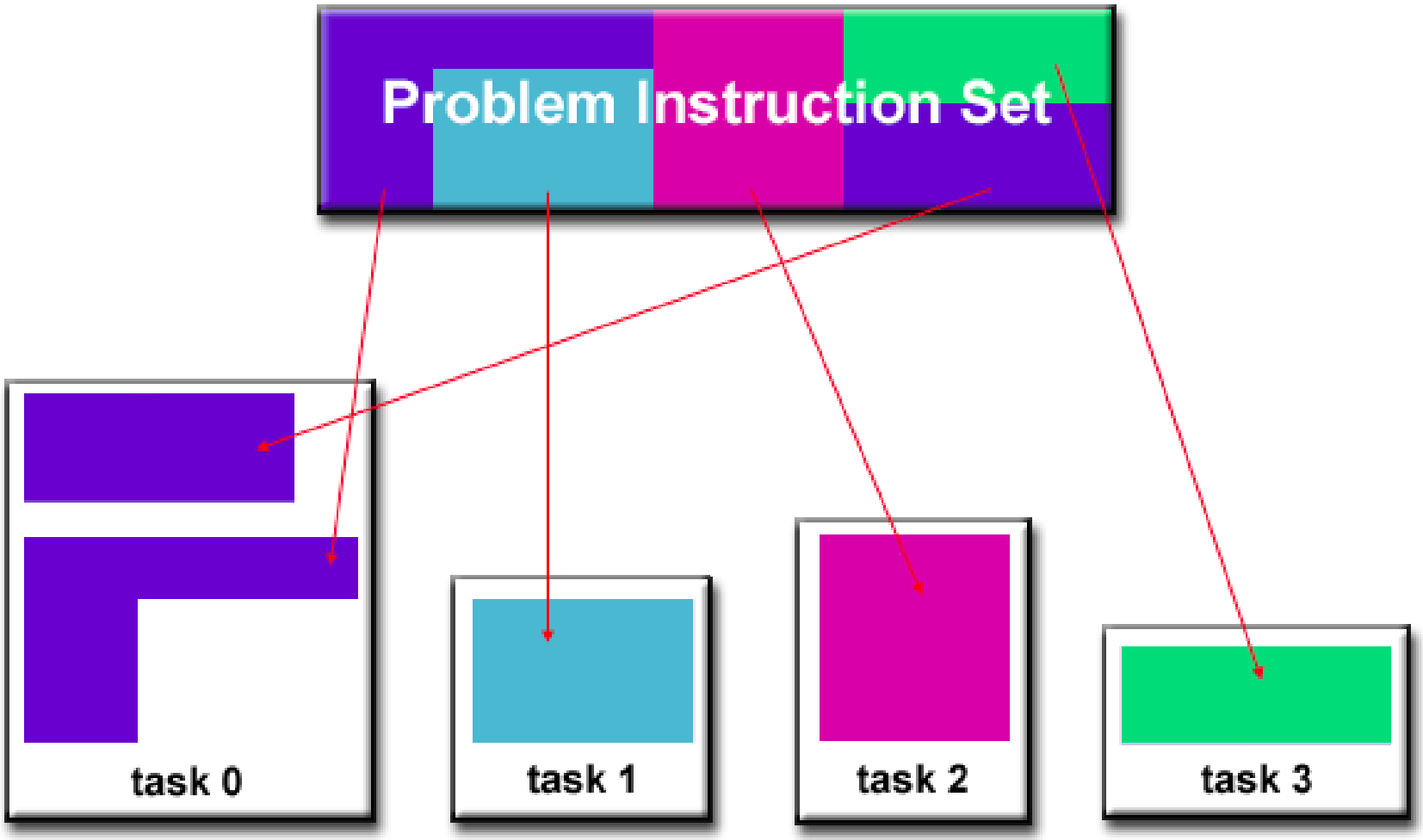
Декомпозиция по данным

Пронумеруем потоки от 1 до N.
Каждый поток P_i получит на обработку количество частей данных:

$$P_i = M/N + \begin{cases} 1, & \text{если } i <= M\%N \\ 0, & \text{в обратном случае} \end{cases}$$

- N – число потоков
- M – число частей данных

Функциональная декомпозиция



Outline

Атомарная операция

Атомарная операция — операция, которая либо выполняется целиком, либо не выполняется вовсе; операция, которая не может быть частично выполнена и частично не выполнена.

В C++ существуют отдельные типы данных `atomic`$_x$*xxx*`atomic`

Атомарная операция

- Атомарность операции можно проверить записав её в виде ассемблерного кода
- Если операция будет представлена одной ассемблерной операцией, то она атомарна

Атомарная операция

- Атомарность операции можно проверить записав её в виде ассемблерного кода
- Если операция будет представлена одной ассемблерной операцией, то она атомарна

- Является ли операция присваивания атомарной?
- Является ли операция инкремента атомарной?
- Является ли операция добавления (для числовой переменной)?

- Атомарность операции может быть гарантирована и программным способом.

Потоковая безопасность

Потоковая безопасность (thread-safety) — это концепция программирования, применимая к многопоточным программам. Код потокобезопасен, если он функционирует исправно при использовании его из нескольких потоков одновременно. В частности, он должен обеспечивать правильный доступ нескольких потоков к разделяемым данным.

Состояние гонки (race condition)

- Состояние гонки = конкуренция
- Однако академически правильный термин для этой проблемы: **неопределённость параллелизма**.

- Проблема: после проверки данных одним потоком, они изменяются другим потоком. Первый поток не замечает этого изменения.

см также проблемы Therac-25

Состояние гонки (race condition)

```
volatile int x;
// Поток 1:
while (!stop) {
  x++;
  ... }

// Поток 2:
while (!stop) {
  if (x%2 == 0)
    System.out.println("x=" + x);
  ... }
```

Пусть x=0. Предположим, что выполнение программы происходит в таком порядке:

1. Оператор if в потоке 2 проверяет x на чётность.
2. Оператор «x++» в потоке 1 увеличивает x на единицу.
3. Оператор вывода в потоке 2 выводит «x=1», хотя, казалось бы, переменная проверена на чётность.

Состояние гонки (race condition)

Решение 1: Локальная копия

```
// Поток 2:
while (!stop)
{
  int cached_x = x;
  if (cached_x%2 == 0)
    System.out.println("x=" + cached_x);
  ...
}
```

Операция копирования переменной должна быть атомарной

Состояние гонки (race condition)

Решение 1: Локальная копия

```
int x;

// Поток 1:
while (!stop){
  synchronized(someObject) {
    x++;}
  ...
}

// Поток 2:
while (!stop){
  synchronized(someObject){
    if (x%2 == 0)
      System.out.println("x=" + x);}
  ...
}
```

Блок synchronized может выполнятся в один момент времени только одним потоком

Взаимная блокировка

Взаимная блокировка (deadlock) – ситуация в многозадачной среде или СУБД, при которой несколько процессов находятся в состоянии ожидания ресурсов, занятых друг другом, и ни один из них не может продолжать свое выполнение.

Шаг	Процесс 1	Процесс 2
0	Хочет захватить A и B, начинает с A	Хочет захватить A и B, начинает с B
1	Захватывает ресурс A	Захватывает ресурс B
2	Ожидает освобождения ресурса B	Ожидает освобождения ресурса A
3	Взаимная блокировка	

Взаимная блокировка

Решение

- Классический способ борьбы с проблемой — разработка иерархии блокировок
- между блокировками устанавливается отношение сравнения и вводится правило о запрете захвата «большей» блокировки в состоянии, когда уже захвачена «меньшая».
- Таким образом, если процессу нужно несколько блокировок, ему нужно всегда начинать с самой «большой» — предварительно освободив все захваченные «меньшие», если такие есть — и затем в нисходящем порядке.

Проблема ABA

- Процесс P_1 читает значение A из разделяемой памяти,
- P_1 вытесняется, позволяя выполняться P_2,
- P_2 меняет значение A на B и обратно на A перед вытеснением,
- $P1$ возобновляет работу, видит, что значение не изменилось, и продолжает…
- Хотя P_1 может продолжать работу, возможно, что его поведение будет неправильным из-за других, скрытых изменений общей памяти (которые он не отслеживал).

Проблема ABA

Решения

- Использование счётчика числа изменений переменной
- атомарная инструкция – сравнение с обменом (compare and swap, CAS)

Outline

Процессы и потоки

- https://en.cppreference.com/w/cpp/thread/thread
- Обёртка над потоками ОС
- github.com/ivtipm/ProcessCalculus/tree/master/examples/example_thread

volatile

- Компилятор оптимизирует код
- В некоторых случаях код может быть существенно переписан
- Например, компилятор может проанализировать код и подставить везде вместо переменной её значение, полагая что переменная не изменяет своего значения
- Однако компилятор не учитывает случая, когда переменная изменяется в отдельном потоке
- Чтобы избежать таких вредных оптимизаций переменную помечают как volatile

- см. также https://en.wikipedia.org/wiki/Volatile_(*computer_programming*)

volatile

```
bool cancel = false;
    while( !cancel ) {
        ;
    }
```

Если cancel не меняется, то ее и проверять каждый раз незачем, компилятор и не будет ее проверять.

Зато если вы укажете перед переменной volatile, то оптимизации не будет. Предполагается, что переменная cancel могла измениться каким-то волшебным образом.

```
volatile bool cancel = false;
while( !cancel ) {
   ;
}
```

Outline

Критическая секция (critical section)

- **Критическая секция** (critical section) – участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком выполнения.

- При нахождении в критической секции двух (или более) потоков возникает состояние «гонки» («состязания»). Для избежания данной ситуации необходимо выполнение четырех условий:
  - Два потока не должны одновременно находиться в критических областях. В программе не должно быть предположений о скорости или количестве процессоров. Поток, находящийся вне критической области, не может блокировать другие потоки. Невозможна ситуация, в которой поток вечно ждет попадания в критическую область.

Критическая секция (critical section)

- При нахождении в критической секции двух (или более) потоков возникает состояние «гонки» («состязания»). Для избежания данной ситуации необходимо выполнение четырех условий:
  - Два потока не должны одновременно находиться в критических областях.
  - В программе не должно быть предположений о скорости или количестве процессоров.
  - Поток, находящийся вне критической области, не может блокировать другие потоки.
  - Невозможна ситуация, в которой поток вечно ждет попадания в критическую область.

Outline

Мьютекс (mutex, от mutual exclusion — «взаимное исключение»)

Outline

Ссылки

Материалы дисциплины
github.com/ivtipm/ProcessCalculus