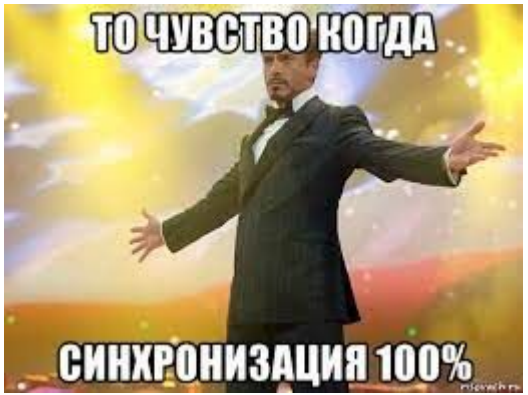
Параллельное программирование

Понятие синхронности;

Синхронность подразумевает, что выполнение операций происходит последовательно, в порядке их вызова. Это означает, что каждая операция должна завершиться до того, как начнется следующая. В программировании это часто проявляется в виде блокирующих вызовов функций, когда выполнение текущего потока приостанавливается до завершения вызванной операции.




```
foo1();

foo2();
```

Понятие асинхронность;

Асинхронность позволяет операциям выполняться независимо друг от друга. Асинхронные вызовы не блокируют выполнение текущего потока, и другие операции могут выполняться параллельно или в ожидании завершения асинхронного вызова. В C++ для асинхронного программирования часто используются std::async, std::future и другие подобные механизмы.

```
#include <iostream>
#include <future>

void task1() {
    std::cout << "Starting task1\n";
    std::cout << "Finished task1\n";
}

void task2() {
    std::cout << "Starting task2\n";
    std::cout << "Finished task2\n";
}

int main() {
    auto future1 = std::async(std::launch::async,
task1);
    auto future2 = std::async(std::launch::async,
task2);
    future1.get();
    future2.get();
    return 0;
}
```

Введение;

Параллельное программирование позволяет существенно ускорить выполнение программ за счет выполнения нескольких операций одновременно. Это особенно важно для задач, требующих значительных вычислительных ресурсов, таких как обработка больших данных, научные вычисления и моделирование. Основные аспекты и технологии параллельного программирования включают следующие понятия и инструменты.

Сравнение понятий;

Характеристика	**Параллелизм**	**Асинхронность**
Цель/Задачи	Увеличение производительности;	Эффективное управление временем;
Ресурсы	Несколько потоков	Один или несколько потоков;
Тип операции	Одновременное выполнение;	Независимое выполнение + ожидание await;
Примеры	OpenMP, MPI, CUDA	std::async,..

Параллелизм.

Параллелизм - выполнение нескольких вычислительных операций одновременно. Параллелизм может быть:

- **Уровня задач**: выполнение различных задач параллельно.
- **Уровня данных**: одновременная обработка различных частей данных.
- **Уровня инструкций**: выполнение нескольких инструкций одновременно в одном потоке выполнения.

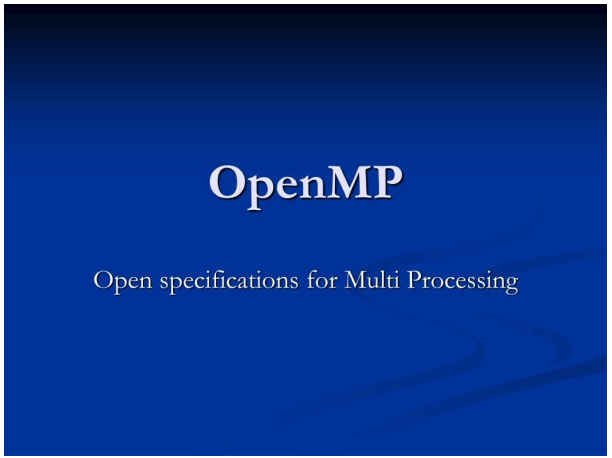
Многопоточность (multithreading) - создание нескольких потоков выполнения в одной программе, которые могут выполняться параллельно.

Распределенные вычисления - использование нескольких компьютеров для выполнения одной задачи.

OpenMP.

Введение;

OpenMP (Open Multi-Processing) — это API, которое поддерживает многопоточное программирование в C, C++ и Fortran. Оно позволяет использовать директивы компилятора для управления параллелизмом. Основные директивы OpenMP начинаются с **#pragma omp.**



Библиотека: `#include <omp.h>`

Compiler command: g++ -fopenmp program.cpp -o program

OpenMP Общий план работы.

1. **Инициализация параллельной секции**: Используйте директиву #pragma omp parallel для создания параллельной секции.

```
#pragma omp parallel
{// Параллельно выполняемый код }
```

2. **Разделение работы**: Используйте директивы, такие как #pragma omp for для параллельного выполнения циклов.

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
   // Параллельно выполняемый код
}
```

3. **Синхронизация**: Управляйте синхронизацией потоков с помощью директив, таких как #pragma omp critical, #pragma omp barrier.
4. **Компиляция**: Скомпилируйте программу с флагом -fopenmp.

#pragma виды;

- **#pragma omp parallel**
  - Начинает параллельный регион, в котором создается команда потоков для выполнения кода.
- **#pragma omp for**
  - Распределяет итерации цикла среди потоков в параллельном регионе.
- **#pragma omp sections**
  - Делит код на независимые секции, которые могут выполняться параллельно.
- **#pragma omp single**
  - Указывает, что блок кода должен выполняться только одним потоком.
- **#pragma omp master**
  - Указывает, что блок кода должен выполняться только основным (главным) потоком.
- **#pragma omp critical**
  - Обозначает критическую секцию, которая должна выполняться только одним потоком в момент времени.
- **#pragma omp barrier**
  - Заставляет все потоки ожидать выполнения друг друга в этой точке.

- **#pragma omp task**
  - Определяет задачу, которая может быть выполнена параллельно с другими задачами.
- **#pragma omp taskwait**
  - Заставляет поток ждать завершения всех задач, порожденных текущей задачей.

reduction.

Директива reduction в OpenMP используется для выполнения операции сокращения (reduction) на переменных. Эта директива позволяет безопасно и эффективно выполнять операции на переменных, к которым обращаются из разных потоков. Операция сокращения объединяет значения частных копий переменной из всех потоков в одну итоговую переменную.

```
#pragma omp parallel for reduction(operator:variable)
```

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < size; ++i) {
    // тут сумма чисел и тп
}
```

операции сокращений;

OpenMP поддерживает несколько операций сокращения:

- \+ (суммирование)
- * (умножение)
- - (вычитание)
- & (побитовое AND)
- | (побитовое OR)
- ^ (побитовое XOR)
- && (логическое AND)
- || (логическое OR)

```
#pragma omp parallel for reduction(*:product)
```

```
#pragma omp parallel for reduction(min:min_val)
```

Пример-1;

```
#include <omp.h>
#include <iostream>

int main() {
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num(); // получение номера каждого потока
        std::cout << "Hello, World! from thread "  << thread_id << std::endl;
    }

    return 0;
}
```

Пример-2;

```
int main() {
   int var1 = 10, var2 = 20, result = 0;
   #pragma omp parallel for reduction(+:result)
   for (int i = 0; i < 2; ++i) {
       // Параллельное сложение var1 и var2
       if (i == 0) {
           result += var1;
       } else {
           result += var2;
       }
   }
   std::cout << "Result: " << result << std::endl;
   return 0;
}
```

OpenMP ЗАДАЧИ;

- **Параллельное суммирование элементов массива** Напишите программу, которая суммирует элементы большого массива, используя OpenMP для параллельного выполнения.
- **Параллельный поиск максимума в массиве** Напишите программу, которая находит максимальный элемент в массиве, используя OpenMP для параллельного выполнения.
- **Параллельная сортировка пузырьком** Напишите программу, которая реализует сортировку пузырьком с использованием OpenMP для параллельной обработки частей массива.

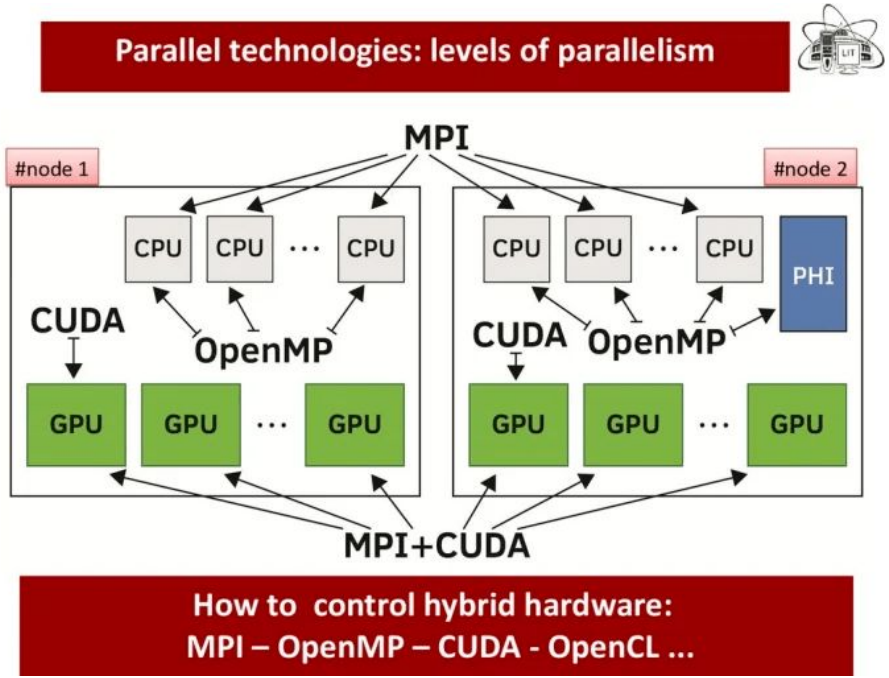
MPI.

Введение;

MPI — это стандарт для программирования распределенных систем, который позволяет процессам обмениваться сообщениями.

Начало работы:

- Установите MPI;
- Настройка окружения, IDE.
- Запуска MPI;



MPI. Концепции/Описание.

- Инициализация и завершение;

```
MPI_Init(&argc, &argv);
MPI_Finalize();
```

- Получение ранга процесса и общего числа процессов;

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

- Отправка и получение сообщений;

```
MPI_Send(&data, count, MPI_INT, dest, tag, MPI_COMM_WORLD);
MPI_Recv(&data, count, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
```

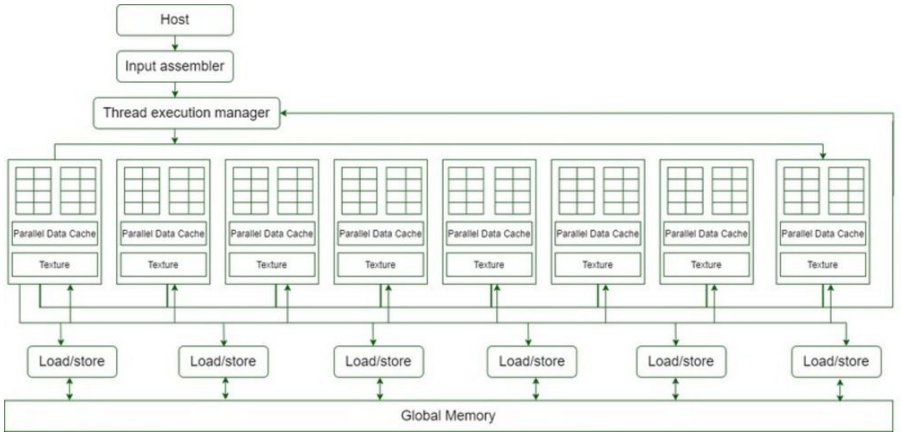
MPI. базовый пример.

```cpp
#include <mpi.h>
#include <iostream>

int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   std::cout << "Hello from process " << rank << " out of " << size << "\n";

   MPI_Finalize();
   return 0;
}
```

CUDA.

Введение.

CUDA — это платформа и API от NVIDIA для разработки программного обеспечения, использующего графические процессоры (GPU).



Понятие кернелы;

кернел - функция, которая выполняется на графическом процессоре (GPU). Она представляет собой код, который вы хотите выполнить параллельно на различных "потоках" в GPU. Каждый поток выполняет ту же функцию, но с разными данными.

Кернелы CUDA запускаются с использованием синтаксиса, особого для CUDA, который позволяет вам указать, сколько потоков должно быть запущено для выполнения кернела и как эти потоки должны быть организованы в блоки. Кернелы могут иметь одну, две или три размерности, что позволяет реализовывать различные структуры параллелизма.

Расчет индекса потока: Поскольку кернел выполняется параллельно на множестве потоков, каждый поток должен знать, какую часть данных он обрабатывает. Для этого используется расчет индекса потока с помощью информации о его идентификаторе блока (blockIdx) и идентификаторе потока (threadIdx). Эти индексы могут быть использованы для доступа к различным элементам данных. Внутри кернела доступ к индексам блока и потока можно получить через blockIdx и threadIdx.

Поскольку кернел выполняется параллельно на множестве потоков, вам нужно обеспечить правильное управление доступом к общей памяти, а также синхронизацию потоков при необходимости.

Кернел пример определения функции;

```
__global__ void fooKernelTest(int *array, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        array[idx] *= 2;
    }
}
```

CUDA. Концепции/Описание;

- **Инициализация и выделение памяти**: Подготовьте память на GPU.

```
int *d_data;
```

```
cudaMalloc((void**)&d_data, size);
```

- **Копирование данных**: Скопируйте данные с CPU на GPU.

```
cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice);
```

- **Написание кернела**: Определите функцию, которая будет выполняться на GPU.
- **Запуск кернела**: Запустите кернел на GPU с указанным количеством блоков и потоков.

```
kernel_function<<<num_blocks, num_threads>>>(d_data);
```

- **Синхронизация и копирование результатов**: Дождитесь завершения выполнения кернела и скопируйте результаты обратно на CPU.

```
cudaDeviceSynchronize ();
cudaMemcpy (h_data, d_data, size, cudaMemcpyDeviceToHost);
```

- **Освобождение памяти;**