Классификация Флинна

Самой ранней и наиболее известной является классификация архитектур вычислительных систем, предложенная в 1966 году Майкл Дж. Флинн (Michael J. Flynn). Классификация 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.

SISD (single instruction stream / single data stream) - одиночный поток команд и одиночный поток данных. К этому классу относятся, прежде всего, классические последовательные машины, или иначе, машины фон-неймановского типа, например, PDP-11 или VAX 11/780. В таких машинах есть только один поток команд, все команды обрабатываются последовательно друг за другом и каждая команда инициирует одну операцию с одним потоком данных. Не имеет значения тот факт, что для увеличения скорости обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка - как машина CDC 6600 со скалярными функциональными устройствами, так и CDC 7600 с конвейерными попадают в этот класс.

MISD (multiple instruction stream / single data stream) - множественный поток команд и одиночный поток данных. Определение подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. Однако ни Флинн, ни другие специалисты в области архитектуры компьютеров до сих пор не смогли представить убедительный пример реально существующей вычислительной системы, построенной на данном принципе. Ряд исследователей относят конвейерные машины к данному классу, однако это не нашло окончательного признания в научном сообществе.

MIMD (multiple instruction stream / multiple data stream) - множественный поток команд и множественный поток данных. Этот класс предполагает, что в вычислительной системе есть несколько устройств обработки команд, объединенных в единый комплекс и работающих каждый со своим потоком команд и данных.

SIMD – это аббревиатура, соответствующая выражению Single Instruction Stream, Multiple Data (один поток выполнения инструкций, много потоков данных). Термин SIMD предложил Майкл Дж. Флинн (Michael J. Flynn) для обозначения функциональности, позволяющей выполнять одну инструкцию для нескольких «потоков» данных. Обладая потенциальными возможностями параллельных вычислений, SIMD инструкции могут использоваться для повышения скорости вычислений в широком спектре прикладных областей, таких как обработка изображений, звука, сигналов, для векторных и матричных вычислений и т.д.

Закон Амдала

В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого медленного фрагмента. Ускорение выполнения программы за счёт распараллеливания её инструкций на множестве вычислителей ограничено временем, необходимым для выполнения её последовательных инструкций.

Пусть необходимо решить некоторую вычислительную задачу. Предположим, что её алгоритм таков, что доля α от общего объёма вычислений может быть получена только последовательными расчётами, а, соответственно, доля $1 - \alpha$ может быть распараллелена идеально (то есть время вычисления будет обратно пропорционально числу задействованных узлов p). Тогда ускорение, которое может быть получено на вычислительной системе из p процессоров, по сравнению с однопроцессорным решением не будет превышать величины:

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

Таблица показывает, во сколько раз быстрее выполнится программа с долей последовательных вычислений α при использовании p процессоров.

$\alpha \setminus p$	**10**	**100**	**1 000**
0	10	100	1 000
10 %	5,263	9,174	9,910
25 %	3,077	3,883	3,988
40 %	2,174	2,463	2,496

Сетевой закон Амдала

Основной вариант закона Амдала не отражает потерь времени на межпроцессорный обмен сообщениями. Эти потери могут не только снизить ускорение вычислений, но и замедлить вычисления по сравнению с однопроцессорным вариантом. Поэтому необходима некоторая модернизация выражения, где с это коэффициент сетевой деградации вычислений:

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p} + c} \qquad c = \frac{W_c \cdot t_c}{W \cdot t}$$

W_c - количество передач данных

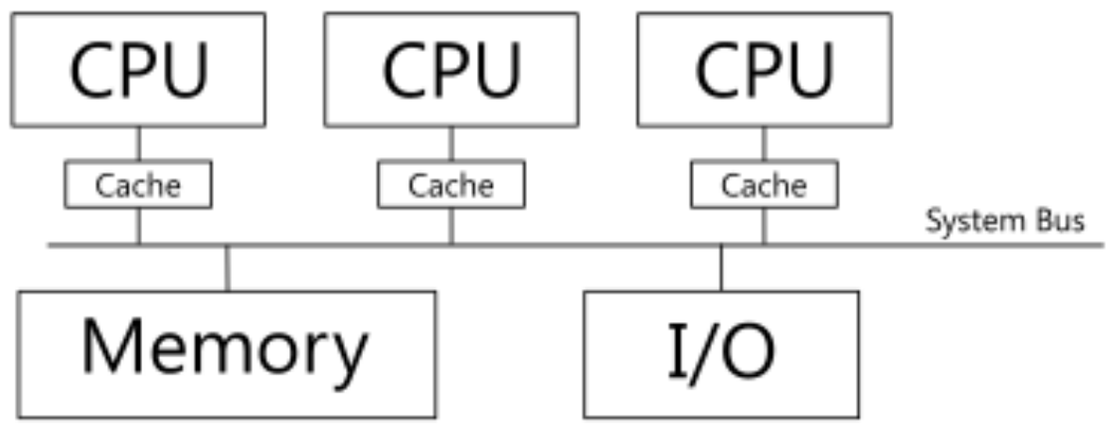
t_c - время одной передачи данных

W - общее число операций

t - время одной операции

Для сетевых расчетов следует учесть, что накладные расходы на передачу данных могут полностью аннулировать прирост производительности!

Многопроцессорная система с общей оперативной памятью или SMP (Symmetric Multiprocessing) позволяет организовать сильно связанный MIMD.



Плюсы:

+ потоки, распределённые по всем CPU

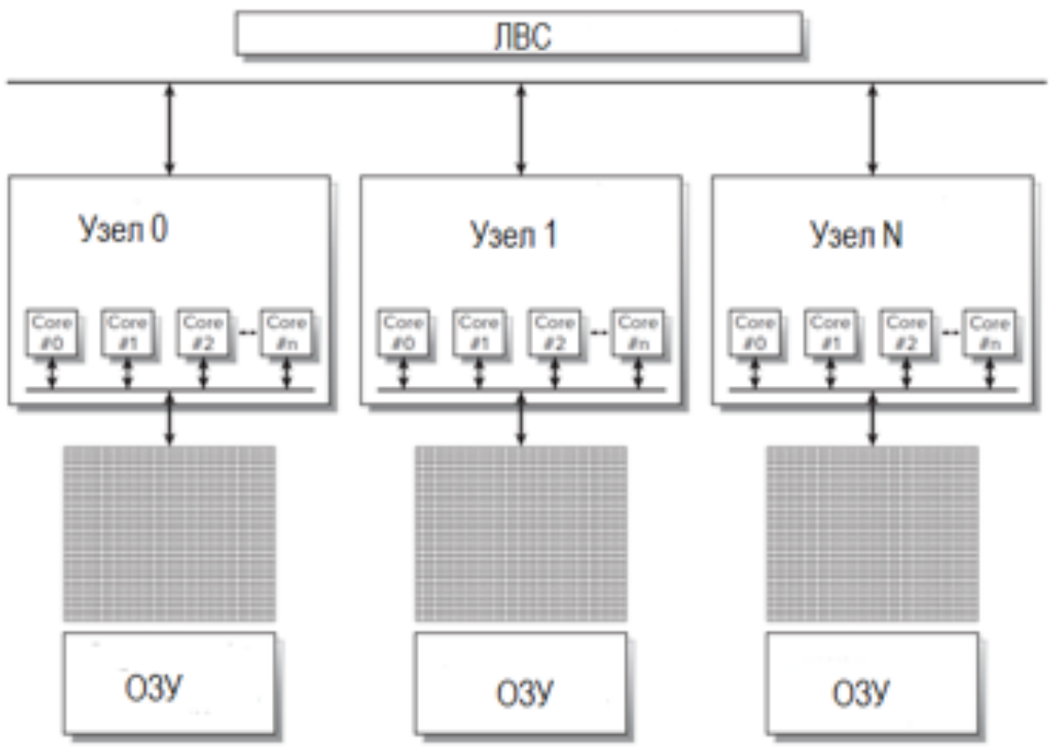
+ отсутствие накладных расходов, связанных с сетью

Минусы:

- необходимость синхронизации

- прирост вычисления ограничен кол-вом CPU

Распределённые системы или MPP (massive parallel processing) позволяет организовать слабо связанный MIMD.



Плюсы:

+ потенциально огромное количество CPU

+ в каждом компьютере может быть сильно связанный MIMD

Минусы:

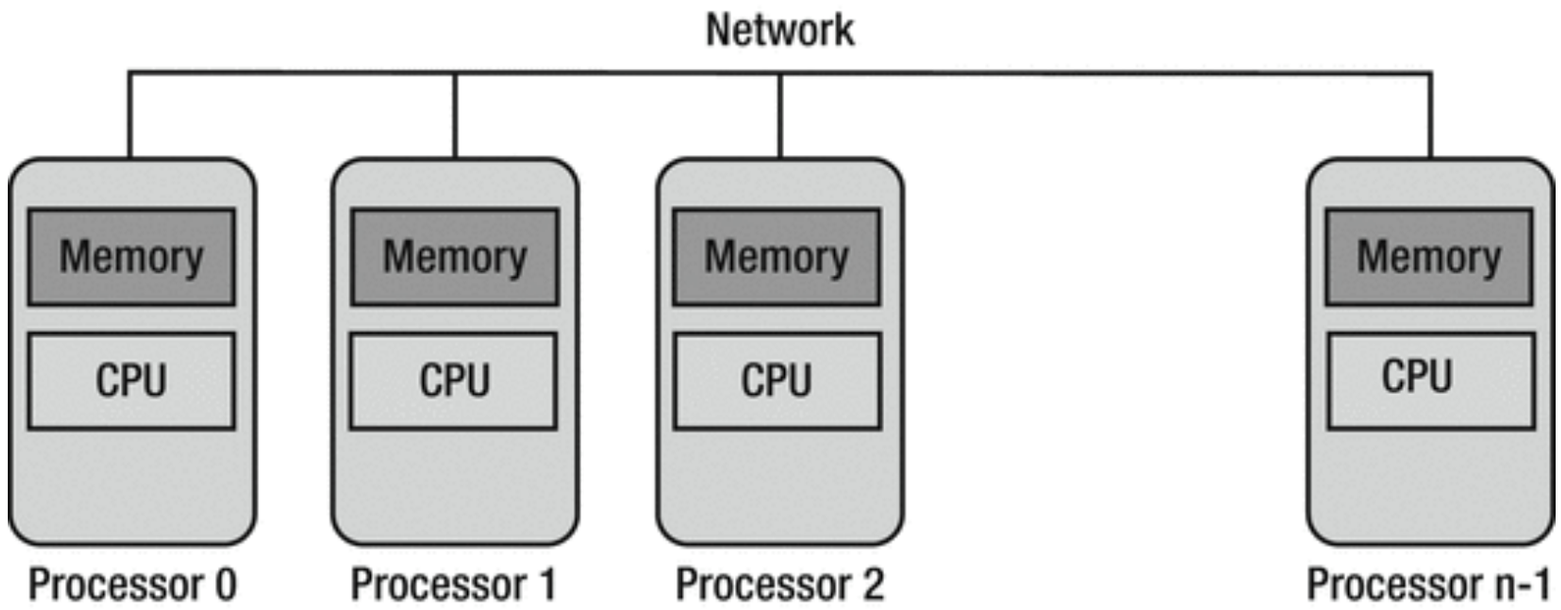
- приходится работать в терминах обмена сообщений

Одним из примеров слабо связанных MIMD вычислений могут служить так называемые добровольные вычисления (volunteer computing). Это распределённые вычисления с использованием предоставленных добровольно вычислительных ресурсов. Современные вычислительные системы для добровольных вычислений строятся на базе грид-систем. Общая схема участия в том или ином проекте распределённых вычислений выглядит так: потенциальный участник загружает клиентскую часть программного обеспечения под свою операционную систему, устанавливает, настраивает и запускает её. Клиент периодически обращается к серверу проекта — запрашивает у него данные для обработки и отправляет результаты.

Необходимые условия распараллеливания для слабо связанных MIMD устройств.

- равномерная загрузка процессов
- минимизация количества и объёма необходимых пересылок данных
- сокращение времени пересылки данных

MPI (Message Passing Interface)



MPI это стандартизованный механизм для построения параллельных программ в модели обмена сообщениями. Стандарт разрабатывается объединением MPI Forum.

Возможности MPI

- Обеспечение удобной абстракции над протоколами сетевого взаимодействия

- Логическая организация кластера, а именно организация виртуальной топологичсекой сети поверх физических протоколов

- Функции синхронизации и коммуникации для параллельных операций

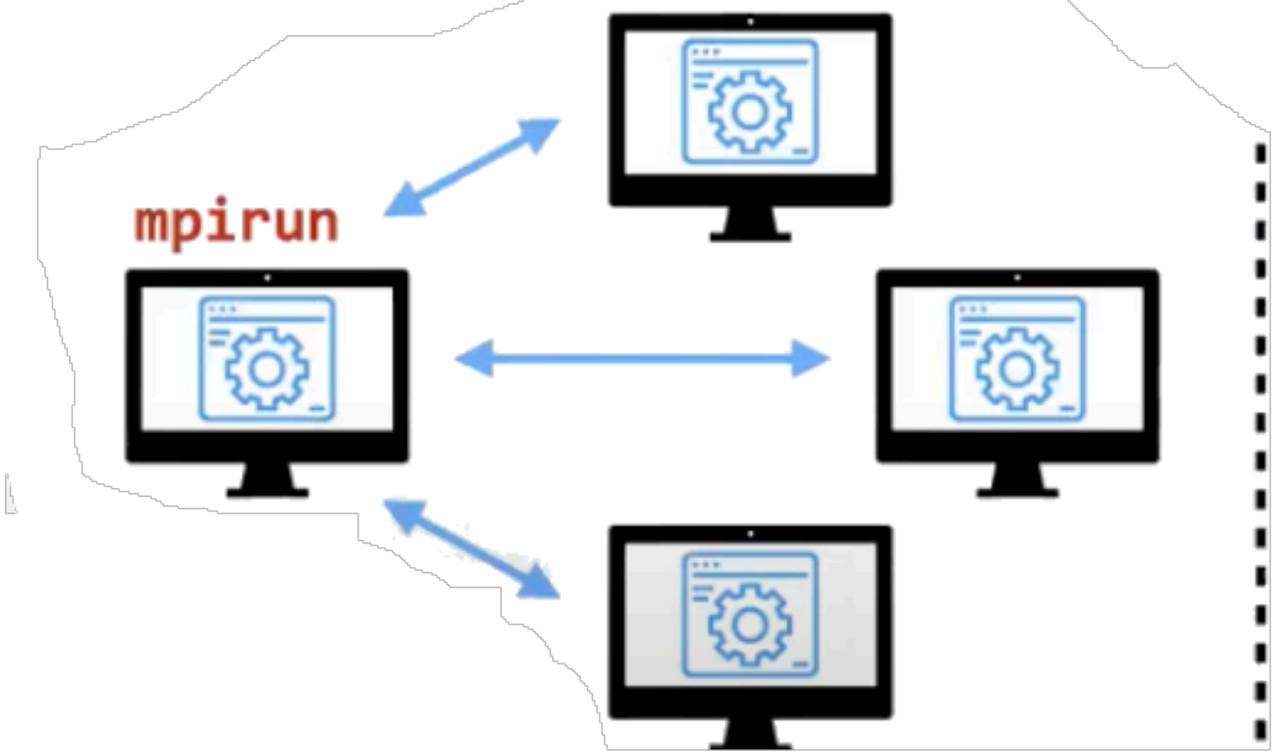
- Кроссплатформенная реализация стандарта

- Биндинги в разные языки программирования

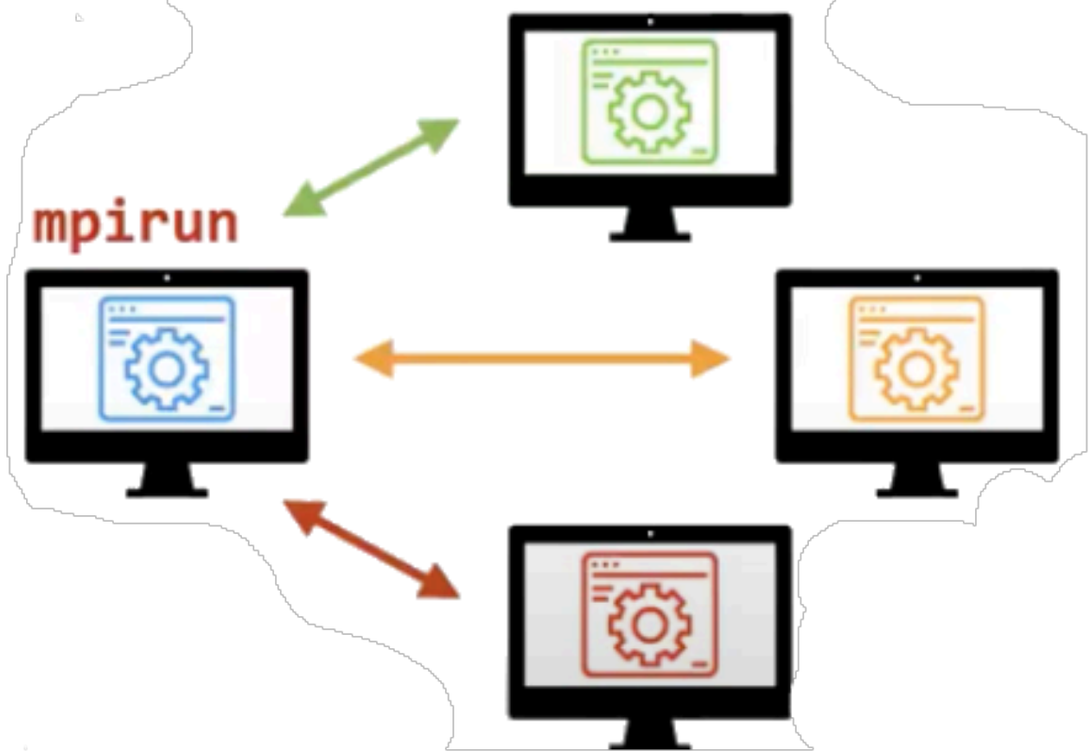
- Ориентированность на слабосвязанный MIMD, однако MPI пригоден для сильносвязанного MIMD при организации многопроцессности

- Новые версии стандарта поддерживают гибридные архитектуры с общей памятью

Single Program Multiple Data (SPMD)



Multiple Programs Multiple Data (MPMD)



MPI содержит в своем составе mpd - сервисный процесс, выполняющийся на узлах и обеспечивающий поддержек виртуальной топологии. Может быть сконфигурирован под конкретную задачу. Также в MPI входит утилита mpirun, которая является менеджером запуска, обеспечивающий корректный старт программ и распределение их экземпляров по вычислительным узлам. Запуск на локальной машине не требует дополнительных настроек mpirun и mpd.

Версии стандарта MPI

MPI 1 - 1994 год.

MPI 1.1 - 12 июля 1995 года.

- передача и получение сообщений между отдельными процессами;
- коллективные взаимодействия процессов;
- взаимодействия в группах процессов;
- реализация топологий процессов;

MPI 2.0 - 18 июля 1997 года.

- динамическое порождение процессов и управление процессами;
- односторонние коммуникации;
- параллельный ввод и вывод;
- расширенные коллективные операции.

MPI 2.1 - 1 сентябрь 2008 года.

MPI 2.2 - 4 сентябрь 2009 года.

MPI 3.0 - 21 сентябрь 2012 года.

MPI 3.1 - 4 июня 2015 года

Реализации MPI

- MPICH2 (Open source, Argone NL)
- MVAPICH2
- IBM MPI
- Cray MPI
- Intel MPI
- HP MPI
- SiCortex MPI
- Open MPI (Open source, BSD License)
- Oracle MPI
- MPJ Express - MPI реализация для Java
- WMPI - реализация для Windows

SPMD (Single Program, Multiple Data)

```
#include <stdio.h>
#include "mpi.h"
#define MAX 100

int main(int argc, char **argv)
{
   int rank, size, n, i, ibeg, iend;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   n=(MAX-1)/size+1;
   ibeg=rank*n+1;
   iend=(rank+1)*n;
   for(i=ibeg; i<=((iend>MAX)?MAX:iend); i++)
      printf ("process %d, %d^2=%d\n", rank, i,
i*i);
   MPI_Finalize();
}
```