УНИВЕРСИТЕТ ИТМО

Parallel algorithms for the analysis and synthesis of data

Sokhin Timur, PhD student

Intro

History

The history of parallel computing can be traced back to the 1960s

- We can mention D825 Modular Data Processing System (1962) as a SMP or ILLIAC IV (1964) as MPP systems.
- 1966 - Flynn creates a taxonomy of architecture (SISD, SIMD, MISD, MIMD)

https://en.wikipedia.org/wiki/ILLIAC_IV#/media/File:ILLIAC_4_parallel_computer.jpg

https://parallel.ru/history/wilson_history.html

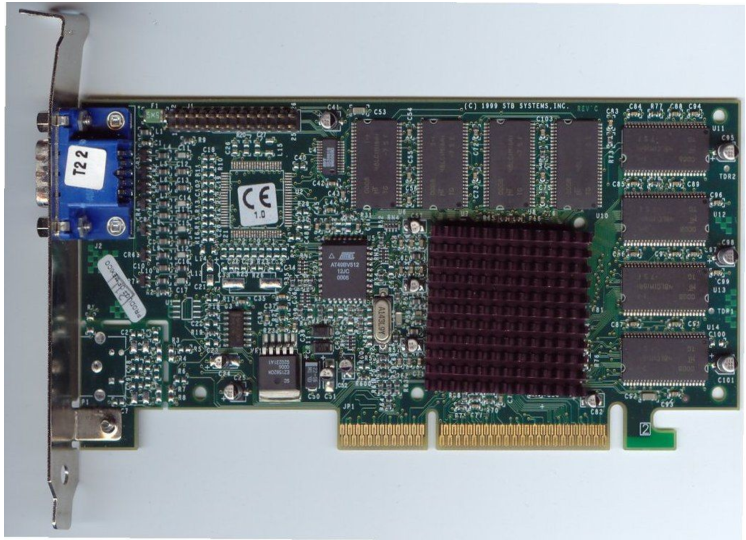
History

- Cray-1 - example of vector processor based computer
- First mentioning of remote procedure call (RPC) in 1978 by Per Brinch Hansen
- 1982 - Cray X-MP with shared memory architecture

https://parallel.ru/history/wilson_history.html

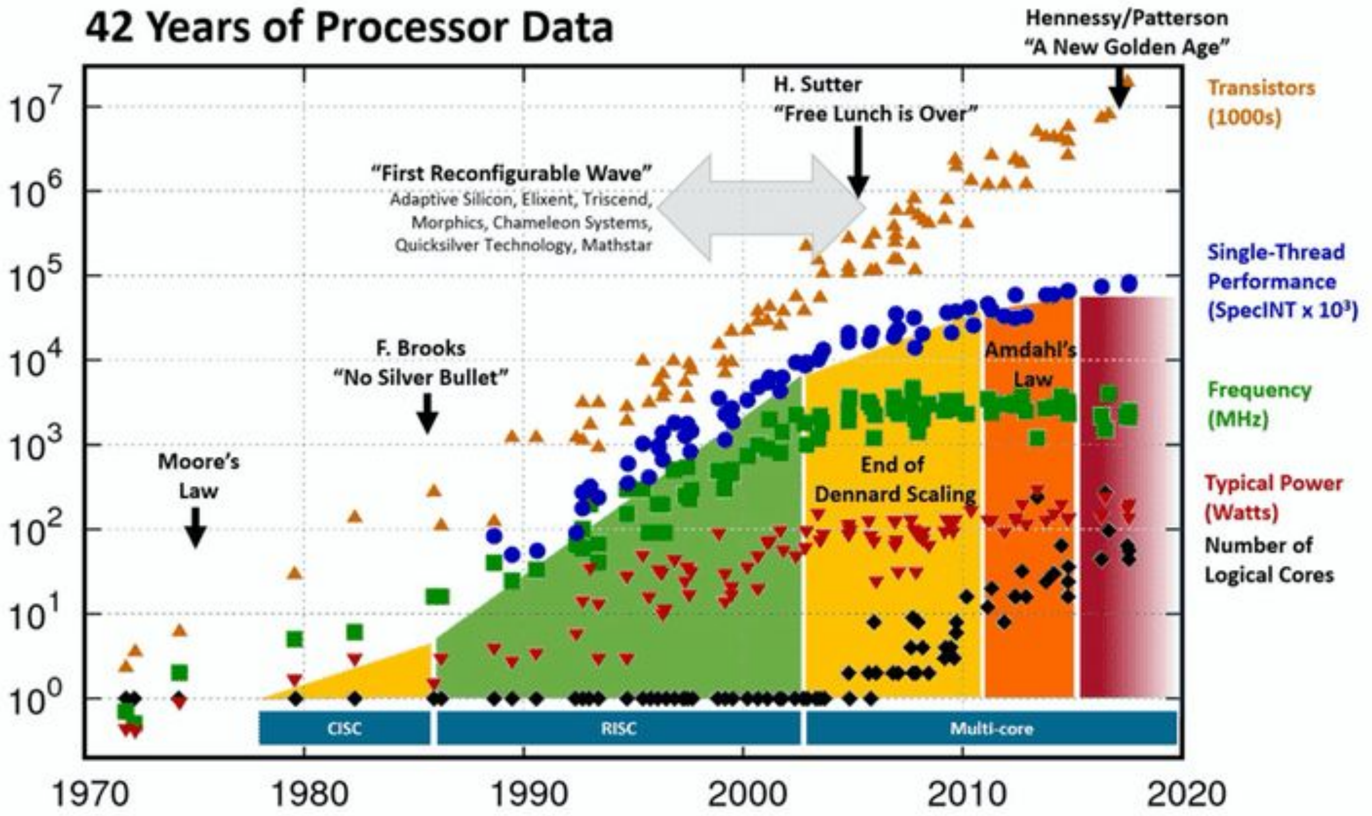
History

- 1993-94 MPI protocol
- 1997 OpenMP protocol
- 2007 CUDA platform



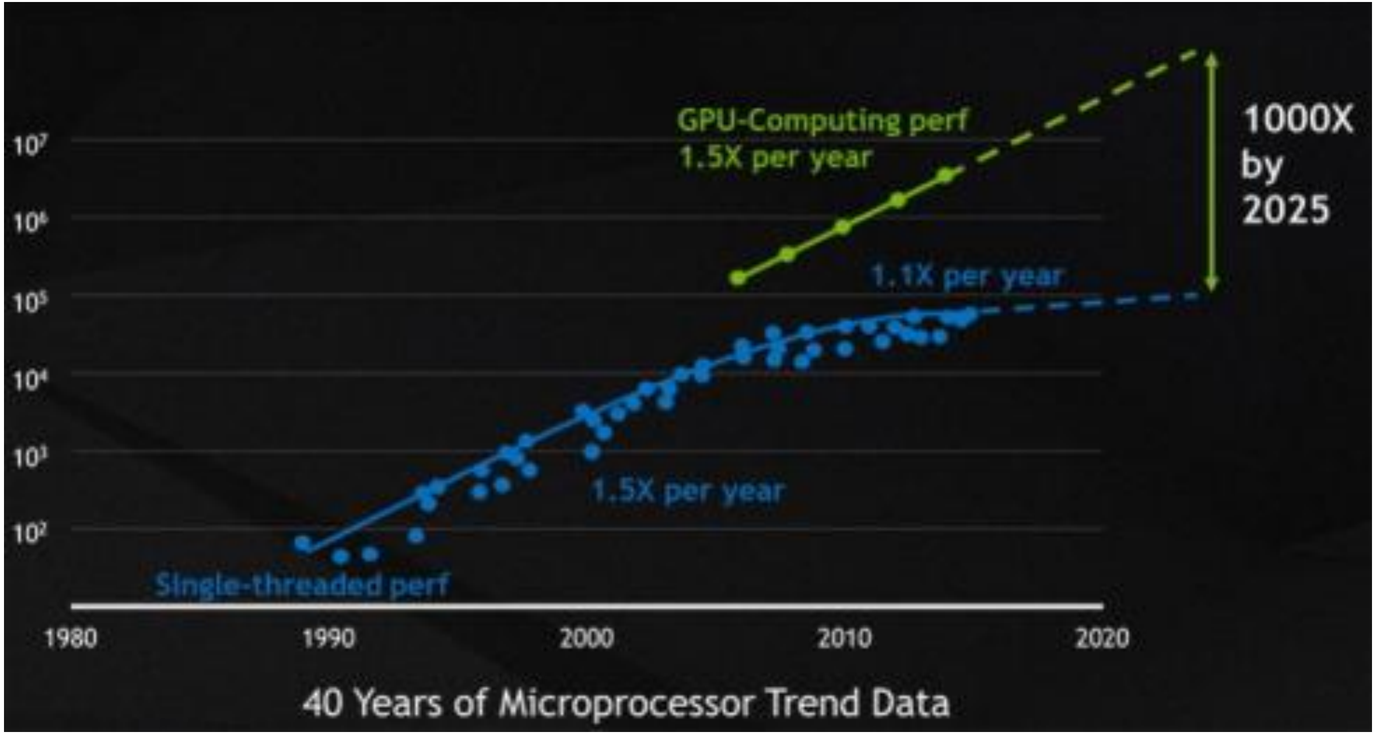
https://parallel.ru/history/wilson_history.html

Reasons



Hennessy and Patterson, Turing Lecture 2018, overlaid over "42 Years of Processors Data"
https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/;"First Wave" added by Les Wilson, Frank Schirrmeister
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

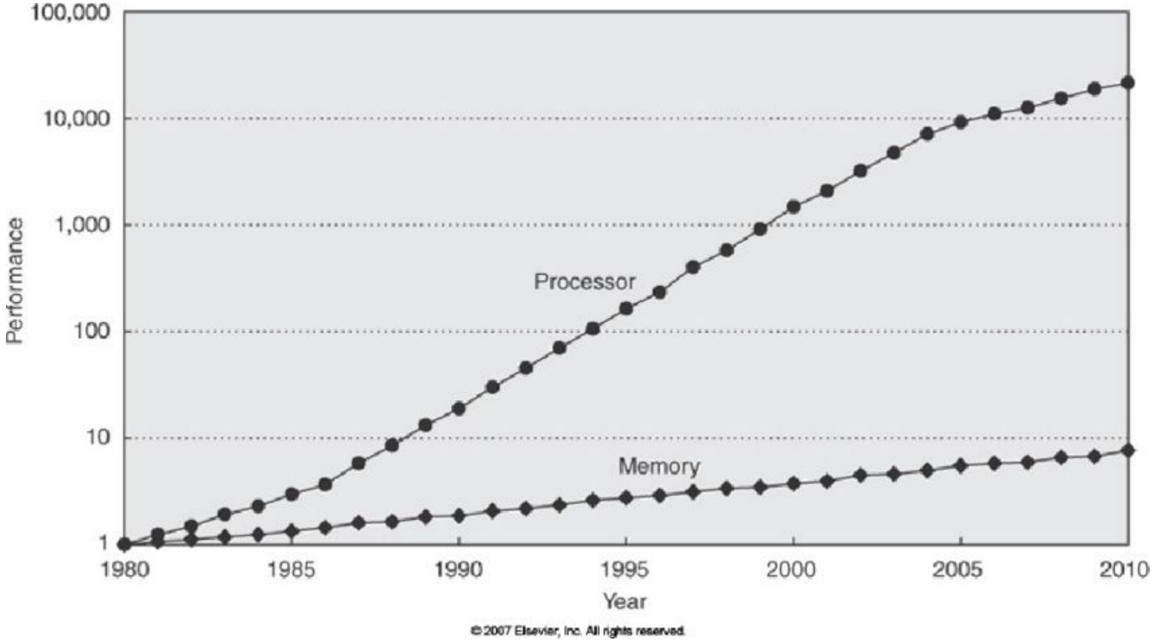
GPGPU


GPU-Computing perf
1.5X per year
1000X
by
2025
1.1X per year
1.5X per year
Single-threaded perf
10^7
10^6
10^5
10^4
10^3
10^2
1980
1990
2000
2010
2020
40 Years of Microprocessor Trend Data

Performance

УНИВЕРСИТЕТ ИТМО

Performance doesn't just depend on computing power and degree of parallelism - don't forget about how memory, networking mechanisms, etc. function.



Locality of reference

Temporal locality: If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future.
Spatial locality: If a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future.

?

```
for i in 0..n
  for j in 0..m
    for k in 0..p
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

```
for i in 0..n
  for k in 0..m
    for j in 0..p
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Locality of reference

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{pmatrix}$$

```
for i in 0..n
  for j in 0..m
    for k in 0..p
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

```
i=0, j=0, k=0

c[0][0] = c[0][0] + a[0][0] * b[0][0];

i=0, j=0, k=1

c[0][0] = c[0][0] + a[0][1] * b[1][0];
```

Memory

$$[a_{00}, a_{01}, a_{02}, a_{10}, a_{11}, a_{12}, a_{20}, a_{21}, a_{22}]$$

$$[b_{00}, b_{01}, b_{02}, b_{10}, b_{11}, b_{12}, b_{20}, b_{21}, b_{22}]$$

$$[c_{00}, c_{01}, c_{02}, c_{10}, c_{11}, c_{12}, a_{20}, a_{21}, c_{22}]$$

Registers

$$[a_{00}, a_{01}, a_{02}, b_{00}, b_{01}, b_{02}, c_{00}, c_{01}, c_{02}]$$

Locality of reference

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{pmatrix}$$

```
for i in 0..n
  for k in 0..m
    for j in 0..p
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Memory

$$[a_{00}, a_{01}, a_{02}, a_{10}, a_{11}, a_{12}, a_{20}, a_{21}, a_{22}]$$

$$[b_{00}, b_{01}, b_{02}, b_{10}, b_{11}, b_{12}, b_{20}, b_{21}, b_{22}]$$

$$[c_{00}, c_{01}, c_{02}, c_{10}, c_{11}, c_{12}, a_{20}, a_{21}, c_{22}]$$

```
i=0, j=0, k=0

c[0][0] = c[0][0] + a[0][0] * b[0][0];

i=0, j=1, k=0

c[0][1] = c[0][1] + a[0][0] * b[0][1];
```

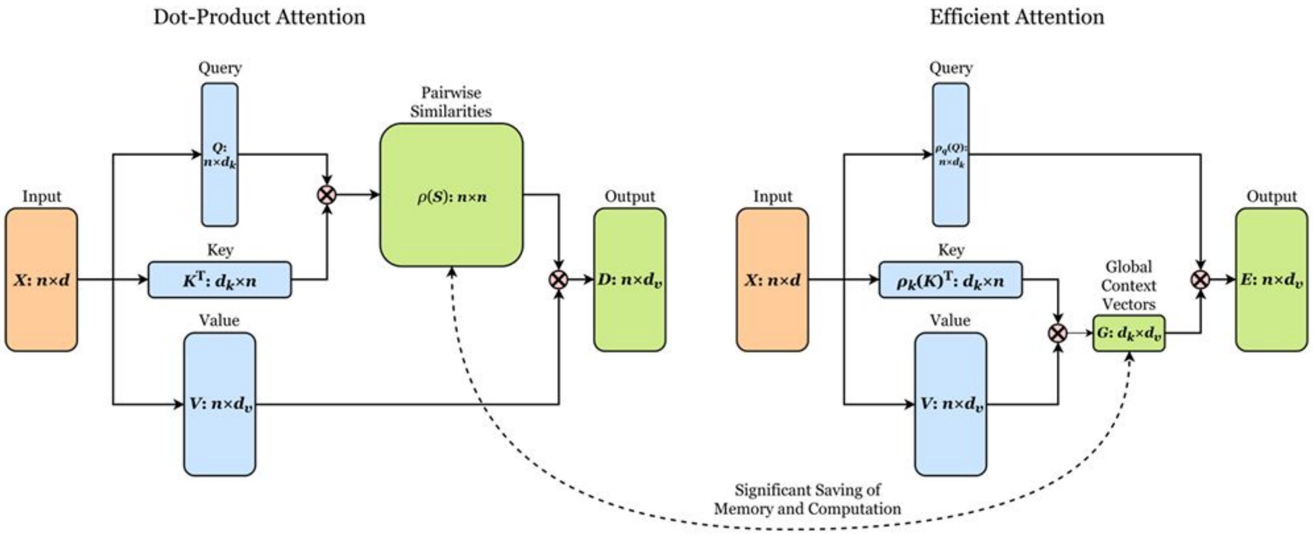
Registers

$$[a_{00}, a_{01}, a_{02}, b_{00}, b_{01}, b_{02}, c_{00}, c_{01}, c_{02}]$$

Computational complexity

УНИВЕРСИТЕТ ИТМО

Transformer networks are well-parallelised, they have an ability to use buffers for inference



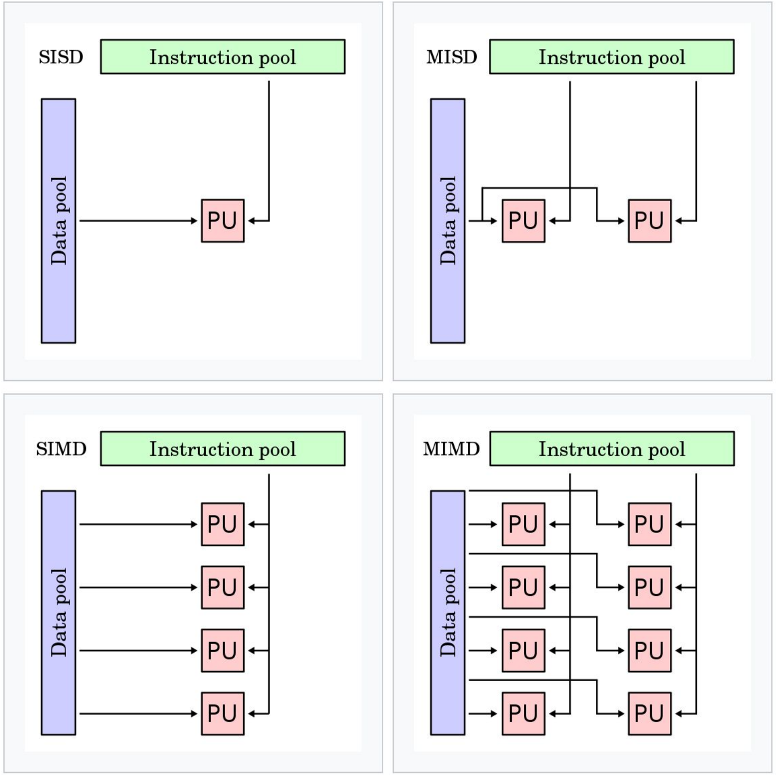
https://arxiv.org/abs/1812.01243

Flynn's taxonomy

single instruction multiple instruction

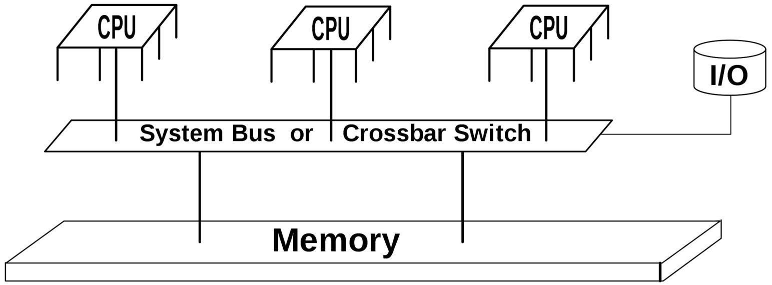
single data

multiple data

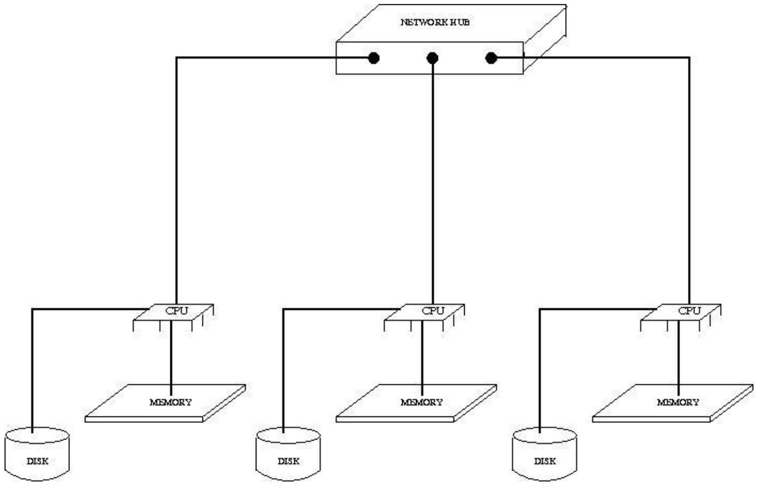


Systematization by access to memory

Shared memory

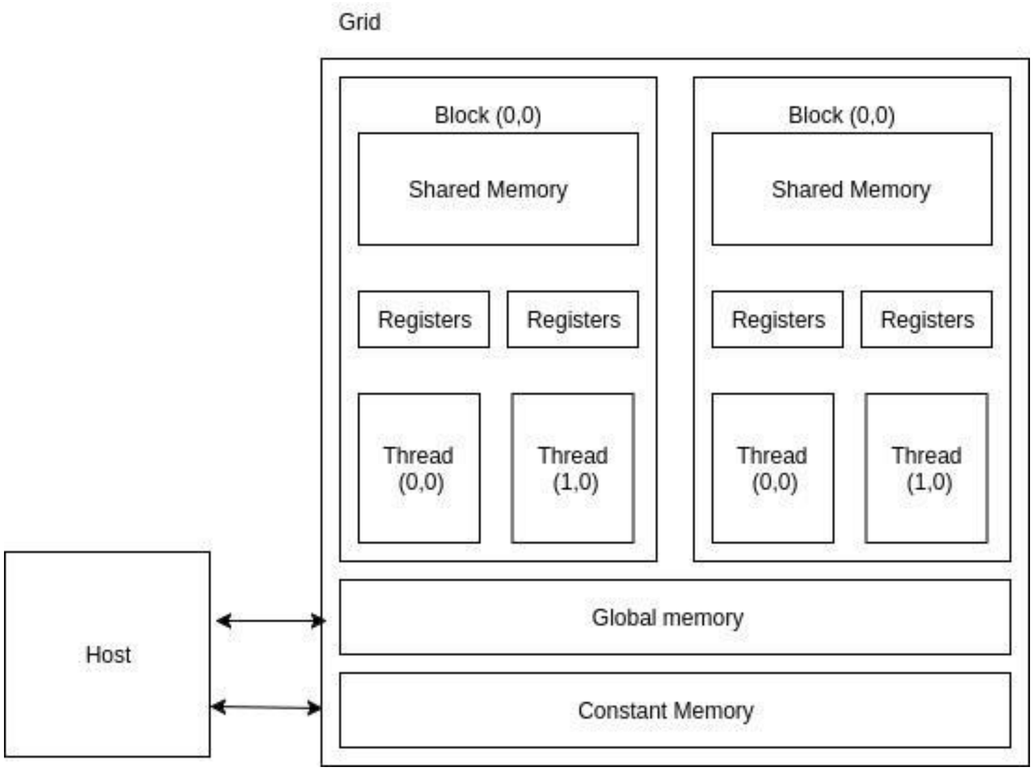


Distributed memory



Systematization by access to memory

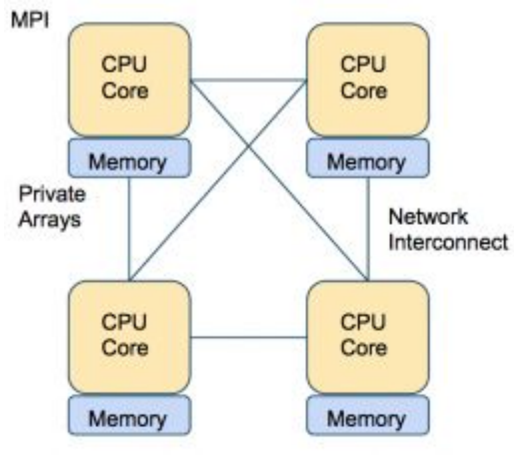
CUDA memory model



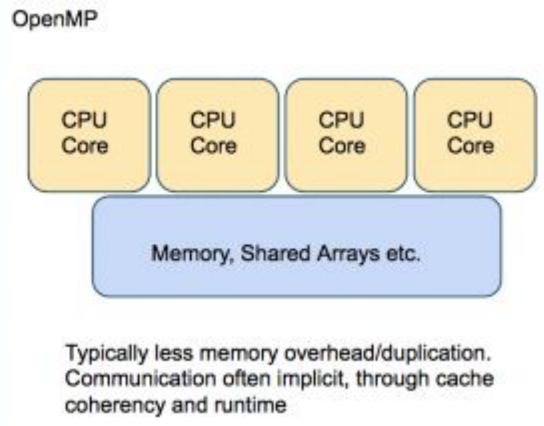
Parallel applications realization

УНИВЕРСИТЕТ ИТМО

Distributed memory



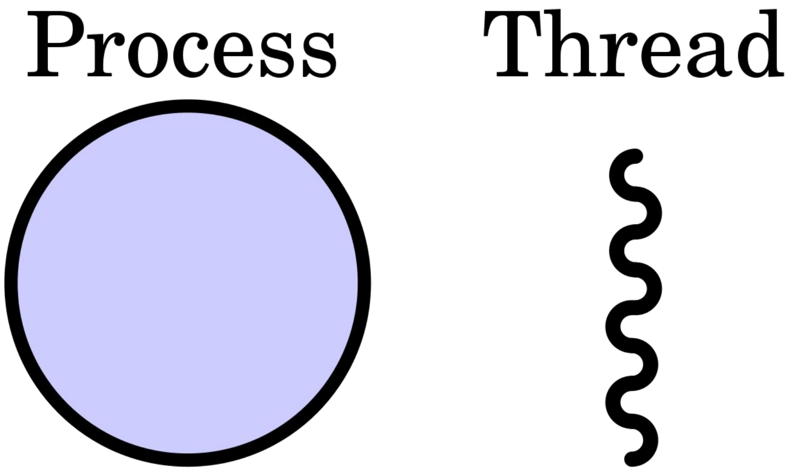
Shared memory



MPI (Message Passing Interface) is a standardized and portable message-passing standard designed to function on parallel computing architectures.

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming.

Process and Thread



Process and Thread

The task is to build a house

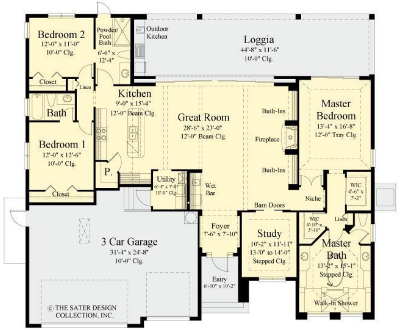
What do you need?

?

Process and Thread

The task is to build a house

What do you need?



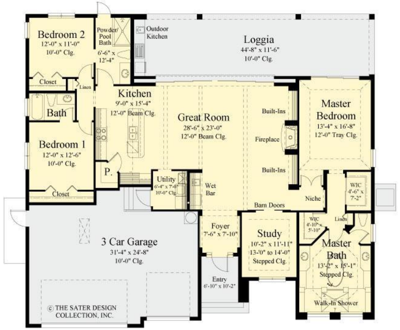
Project

Process and Thread

УНИВЕРСИТЕТ ИТМО

The task is to build a house

What do you need?



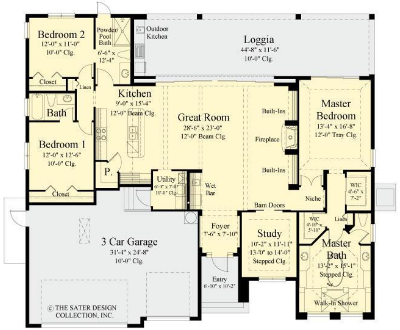
Project

Materials
and instruments

Process and Thread

The task is to build a house

What do you need?



Project

Materials
and instruments

Land

Process and Thread

The task is to build a house

What do you need?

Project

Workers

Materials
and instruments

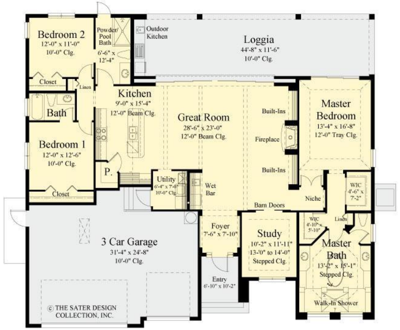
Land

Process and Thread

УНИВЕРСИТЕТ ИТМО

The task is to build a house

What do you need?



Project = Program

All of these is Process

Workers = Threads

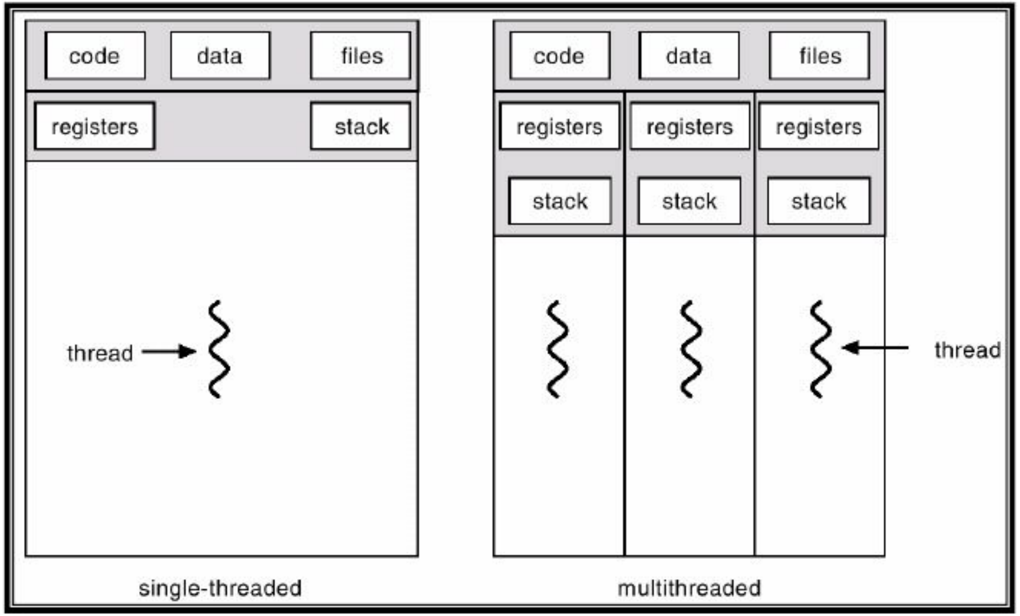
Materials
and instruments = Data

Land = Address space

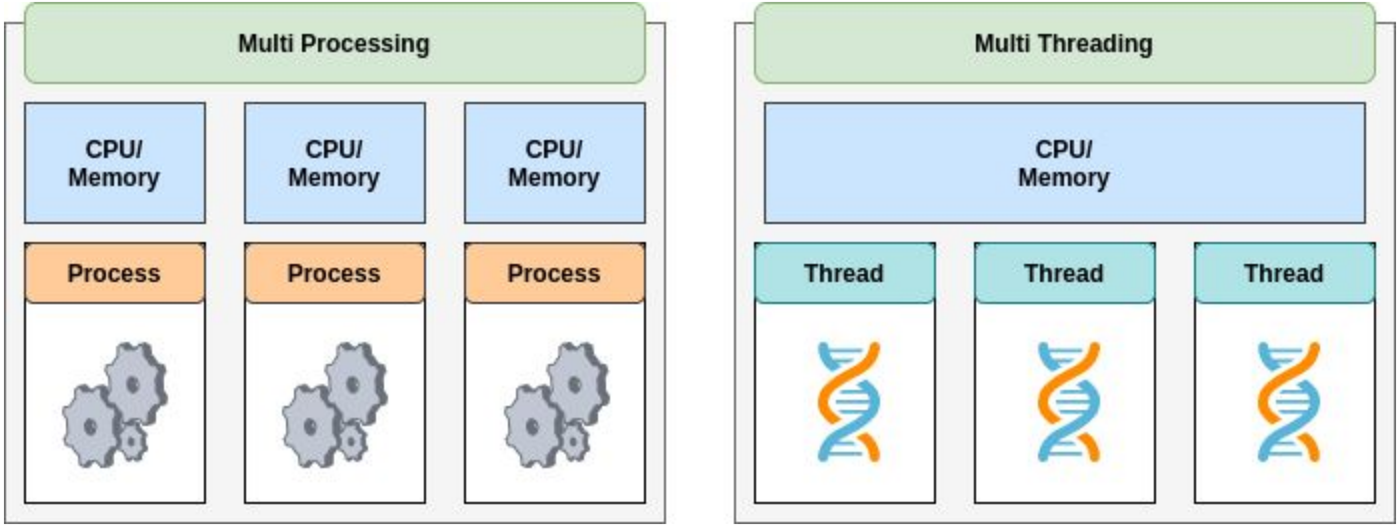
Process and Thread

Process – is a “container” for threads

Processes are isolated from each other, and threads can run in the same process.



Multi Threading



Multi-threaded programming is only possible on shared memory systems.

Algorithm?

- many algorithms can be implemented with parallel computing
- anything we work with - is a data
- it's more about approaches to parallel computing than specific algorithms

MapReduce

Model of parallel computations proposed by Google.

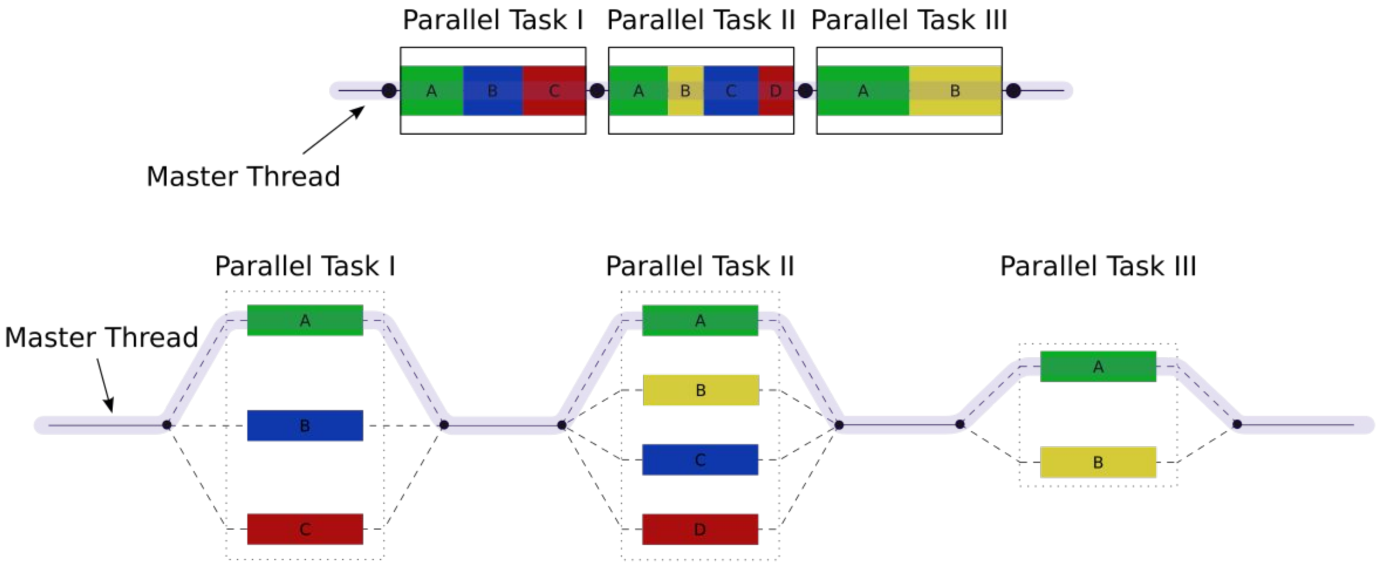
- main idea is to process huge amount of data with thousands machines
- task is splitted on stages:
  - Map - apply simple instruction for each data sample on a local storages
  - Shuffle - reorder data based on keys constructed during map operation in order to keep all data with the same key on a same node
  - Reduce - process each group of reordered data

OpenMP

OpenMP

УНИВЕРСИТЕТ ИТМО

Fork – join parallelism



OpenMP

One version of the program for parallel and sequential execution.

SPMD (Single Program Multiple Data) model of parallel programming: the same code is used for all parallel threads.

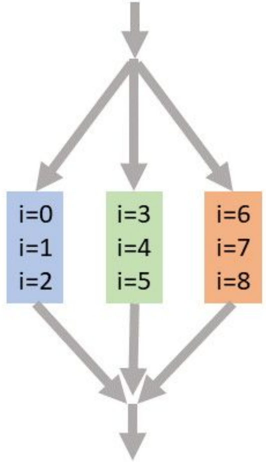
OpenMP includes:

- compiler directives
- helper function
- environment variables

OpenMP

Basics directives embed directly into serial code

```
#pragma omp directive-name (clause((,) clause)...)
```




```
...

#pragma omp parallel
{


#pragma omp for
for (int i = 0; i < 9 ; ++i) {
    a[i] = i;
}


}

...
```

OpenMP

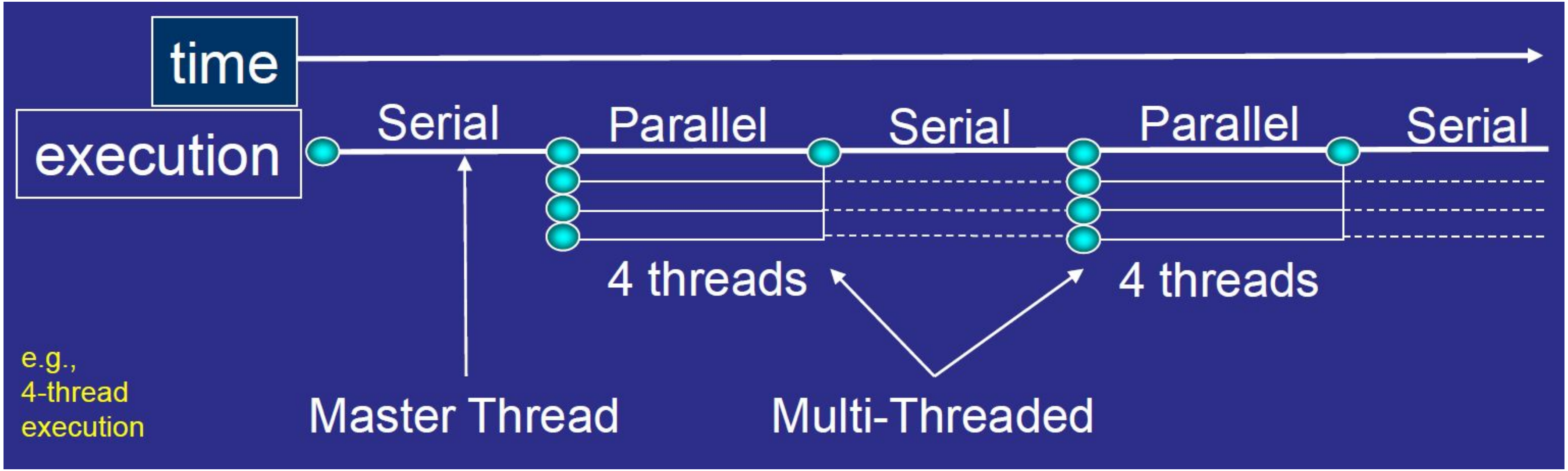
```
#include <omp.h>
#include <iostream>
using namespace std;
int main()
{
    int a[100],b[100],c[100];

    for (int i=0;i<100;i++)
    {
        a[i]=1;
        b[i]=1;
    }

    for (int i=0;i<100;i++)
    {
        c[i]=a[i]+b[i];
    }
}
```

```
#include <omp.h>
#include <iostream>
using namespace std;
int main()
{
    int a[100],b[100],c[100];
    #pragma omp parallel for
    for (int i=0;i<100;i++)
    {
        a[i]=1;
        b[i]=1;
    }
    #pragma omp parallel for
    for (int i=0;i<100;i++)
    {
        c[i]=a[i]+b[i];
    }
}
```

OpenMP

When the program starts, the main thread or master is spawned.

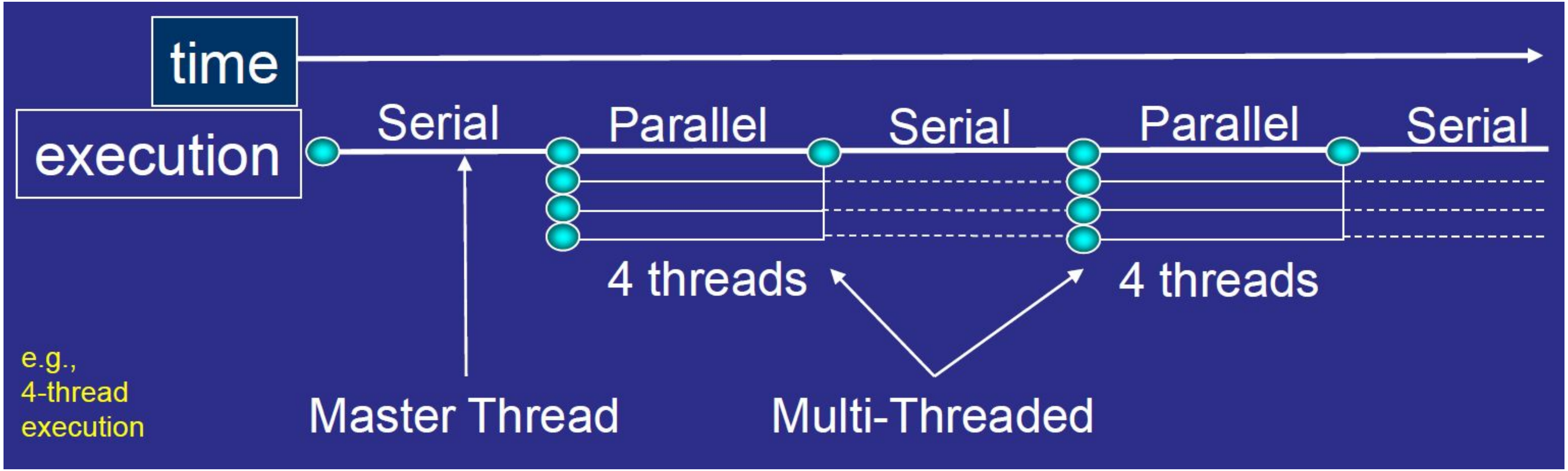
Only the main thread executes all sequential blocks of the program.



OpenMP

For create additional threads for parallel section you should write directive

```
#pragma omp parallel (clause((), clause)…)
{
    \\ parallel code
}
```



OpenMP

For create additional threads for parallel section you should write directive

```
#pragma omp parallel (clause((), clause)…)
{
    \\ parallel code
}
```

```
#include <stdio.h>
#include <locale>

int main()
{
    printf("Serial block 1\n");
#pragma omp parallel
    {
        printf("Parallel block\n");
    }
    printf("Serial block 2\n");
}
```

OpenMP

For create additional threads for parallel section you should write directive

```
#pragma omp parallel num_threads(5)

#pragma omp parallel if (condition)
```

```cpp
#include <iostream>
#include "omp.h"
#include <string>

using namespace std;
int main()
{
    string hw = "Hello, world\n";
#pragma omp parallel num_threads(5)
    {
    cout << hw;
    }
    return 0;
}
```

Memory model. Variable Classes in OpenMP

Shared

Data race

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

using namespace std;

int main()
{
    int x = 0;
#pragma omp parallel shared(x) num_threads(30)
    {
        x += 1;
    }
    cout << "x = " << x << endl;
    return 0;
}
```

Memory model. Variable Classes in OpenMP

Shared

Data race

```cpp
#include <iostream>
#include "omp.h"
#include <string>

using namespace std;

int main()
{
    int x = 0;
#pragma omp parallel shared(x) num_threads(30)
    {
        x += 1;
    }
    cout << "x = " << x << endl;
    return 0;
}
```

Private

```c
#include <stdio.h>
#include <omp.h>
#include <locale>

int main()
{
    int n = 1;

    printf("n in sequential area (start): %d\n", n);

#pragma omp parallel private(n) num_threads(4)
    {
        printf("The value of n in the thread (at the input): %d\n", n);

        n = omp_get_thread_num(); // We assign n the number of the current thread
        printf("The value of n in the thread (at the output): %d\n", n);

    }
    printf("n in sequential area (end): %d\n", n);
    return 0;
}
```

УНИВЕРСИТЕТ ИТМО

private – create local variable for each thread

firstprivate – create local variable for each thread with initialization from previous serial part

```
#include <stdio.h>
#include <omp.h>
#include <locale>

int main()
{
    int n = 1;

    printf("The value of n at the beginning: %d\n", n);
#pragma omp parallel firstprivate(n)
    {
        printf("The value of n in the thread (at the input): %d\n", n);
        n = omp_get_thread_num(); //  assign the variable n to the sequence number of the thread
        printf("The value of n in the thread (at the output): %d\n", n);
    }
    printf("The value of n at the end: %d\n", n);
}
```

Memory model. Variable Classes in OpenMP

УНИВЕРСИТЕТ ИТМО

private – create local variable for each thread

lastprivate – create variable after parallel part with initialization from last parallel section

```
#include <stdio.h>
#include <omp.h>
#include <locale>

int main()
{
    int n = 1;
    int i = 0;
    int a;
    printf("The value of n at the beginning: %d\n", n);
#pragma omp parallel for private(i) lastprivate(a) num_threads(5)
    for(i=0;i<5;i++)
        {
            a=i+1;
            n = omp_get_thread_num(); //  assign the variable n to the sequence number of the thread
            printf("The value of a in the thread %d\n: %d\n",a,n);
        }
printf("The value of a at the end: %d\n", a);
}
```

Modes of execution of multithreaded programs

single

- private
- firstprivate
- copyprivate
- nowait

```
#include <iostream>
#include <stdio.h>
#include <omp.h>
int main()
{
    int num;
    #pragma omp parallel num_threads(4) private(num)
    {
        num = omp_get_thread_num();
        printf("Before the directive single num=%d \n", num);
#pragma omp barrier
#pragma omp single copyprivate(num)
        {
            printf("Enter an integer: ");
            scanf("%d", &num);
        }
        printf("After the directive single num=%d \n", num);
    }
}
```

Modes of execution of multithreaded programs

УНИВЕРСИТЕТ ИТМО

single nowait

```
#include <iostream>
#include <stdio.h>
#include <omp.h>
int main()
{
    double k = 0;
#pragma omp parallel num_threads(4) firstprivate(k)
    {
        printf("Before single without nowait \n");
#pragma omp single
        {
            // This loop is added so that the thread can do some work.
            for (int i = 0; i < 100000; i++)
            {
                k += (double)i / (i + 1);
            }
            printf("In single directive\n");
        }
        printf("After the single directive without nowait. This message will never be earlier than the previous ones. k = %f \n",k);

#pragma omp barrier // This directive synchronizes threads

        printf("Before single directive with nowait \n");
#pragma omp single nowait
        {
            // his loop is added so that the thread can do some work.
            for (int i = 0; i < 100000; i++)
            {
                k += (double)i / (i + 1);
            }
            printf("In single directive\n");
        }
        printf("After the single directive with nowait. This message may be earlier than the previous ones. k =  %f \n", k);
    }
    return 0;
}
```

Modes of execution of multithreaded programs

УНИВЕРСИТЕТ ИТМО

master

```c
#include <stdio.h>
#include <locale>
#include <omp.h>

int main()
{

    int n;
#pragma omp parallel private(n)
    {
        n = 1;
#pragma omp master
        {
            n = 2;
        }
        printf("The first value of the n thread %d: %d\n", omp_get_thread_num(),n);
#pragma omp barrier
#pragma omp master
        {
            n = 3;
        }
        printf("The second value of the n thread %d: %d\n", omp_get_thread_num(),  n);
    }
    return 0;
}
```

Parallelizing loops

```
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
    const long int n = 40000000;
    double* a = new double[n];
    double* b = new double[n];
    double* c = new double[n];
    for (long int i = 0; i < n; i++)
    {
        a[i] = (double)rand() / RAND_MAX;
        b[i] = (double)rand() / RAND_MAX;
    }
    double time = omp_get_wtime();

    for (long int i = 0; i < n; i++)
        {
            c[i] = a[i] + b[i];
        }
    cout << "c[100]=" << c[100] << endl;
    cout << "Time = " << (omp_get_wtime() - time) << endl;
    delete[] a;
    delete[] b;
    delete[] c;
}
```

Parallelizing loops

УНИВЕРСИТЕТ ИТМО

Without omp

```cpp
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
    const long int n = 40000000;
    double* a = new double[n];
    double* b = new double[n];
    double* c = new double[n];
    for (long int i = 0; i < n; i++)
    {
        a[i] = (double)rand() / RAND_MAX;
        b[i] = (double)rand() / RAND_MAX;
    }
    double time = omp_get_wtime();

    for (long int i = 0; i < n; i++)
        {
            c[i] = a[i] + b[i];
        }
    cout << "c[100]=" << c[100] << endl;
    cout << "Time = " << (omp_get_wtime() - time) << endl;
    delete[] a;
    delete[] b;
    delete[] c;
}
```

With omp

```cpp
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
    const long int n = 40000000;
    double* a = new double[n];
    double* b = new double[n];
    double* c = new double[n];
    for (long int i = 0; i < n; i++)
    {
        a[i] = (double)rand() / RAND_MAX;
        b[i] = (double)rand() / RAND_MAX;
    }
    double time = omp_get_wtime();
#pragma omp parallel shared(a,b,c)
    for (long int i = 0; i < n; i++)
        {
            c[i] = a[i] + b[i];
        }
    cout << "c[100]=" << c[100] << endl;
    cout << "Time = " << (omp_get_wtime() - time) << endl;
    delete[] a;
    delete[] b;
    delete[] c;
}
```

Parallelizing loops


```cpp
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
    const long int n = 40000000;
    double* a = new double[n];
    double* b = new double[n];
    double* c = new double[n];
    for (long int i = 0; i < n; i++)
    {
        a[i] = (double)rand() / RAND_MAX;
        b[i] = (double)rand() / RAND_MAX;
    }
    double time = omp_get_wtime();
#pragma omp parallel shared(a,b,c)
    for (long int i = 0; i < n; i++)
        {
            c[i] = a[i] + b[i];
        }
    cout << "c[100]=" << c[100] << endl;
    cout << "Time = " << (omp_get_wtime() - time) << endl;
    delete[] a;
    delete[] b;
    delete[] c;
}
```

With omp, parallel for

```cpp
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
    const long int n = 40000000;
    double* a =new double[n];
    double* b =new double[n];
    double* c =new double[n];
    for (long int i = 0; i < n; i++)
    {
        a[i] = (double)rand() / RAND_MAX;
        b[i] = (double)rand() / RAND_MAX;
    }
    double time = omp_get_wtime();
    for (int j = 0; j < 100; j++)
    {
#pragma omp parallel shared(a,b,c)
#pragma omp for
        for (long int i = 0; i < n; i++)
        {
            c[i] = a[i] + b[i];
        }
    }
    cout << "c[100]=" << c[100] << endl;
    cout << "Time = " << (omp_get_wtime() - time) / 100 << endl;
    delete[] a;
    delete[] b;
    delete[] c;
}
```