# 1.1. Аспекти на производителността, .NET код?

1. **Производителност** - *good results* for the *given expectations*
2. **Изисквания:**
   •**functional** requirements: *what* a software must do
   •**non-functional** requirements *how* the system has to work (**system architecture**):
   Security (сигурност)
   Reliability (надеждност)
   Testability (възможност за автоматично тестване)
   Maintainability (лесна поддръжка)
3. **Performance engineering** – the structure behind the goal to succeed in respecting all the nonfunctional requirements that a software development team should respect
   •Objectives
   I. Reducing software maintenance costs.
   II. Increasing business revenue.
   III. Reducing hardware acquisition costs.
   IV. Reducing system rework for performance issues.
4. **Аспекти на производителността:**
   •Latency (време за реакция) - the time between a request and response;
   •Throughput (пропускателна способност) - The **speed rate** of anything is the main task of the given product or function being valued;
   •Resource usage (използвани ресурси) - Memory usage –primary concern. No leaks! ;
   •Availability/reliability (достъпност/надеждност) - the software being in up-time, actually running, without issues in any condition. The more a system is available, the more it is reliable
   •Scalability (мащабируемост) - the ability of a single function or entire application to boost its performance—as the number of processors rise or the number of servers increases
   •Efficiency (ефективност) - Avoid wasting any computational power and consequently, electrical power (critical in mobile computing, where battery life is never enough)
5. **Performance aspects of a desktop app**
   •The main performance aspect composing a requisite for a desktop application is **latency;**
   •*To whom is this application going to serve?*
   •Human users react as explained in the following bullet list:
   ◦100 milliseconds is the time limit to make sure an application is actually reacting well
   ◦1 second is the time limit to bring users to the application workflow, otherwise users will experience delay, 10 seconds is the time limit to keep the users' attention on the given
   •Low resource usage is another key aspect for a desktop application performance requisite
   •Power, Availability, Scalability – not a problem for a desktop app
6. **Performance aspects of a mobile app**
   •key performance aspect is **resource usage**
   • Latency - overshadowed by the system architecture.
7. **Performance aspects of a server app**
   •the focus is on **throughput (**the ability to process as many transactions the workflow or scheduler can process)
   •Latency –no user interaction. •Resource usage is also sensible (imagine a server crash)
   •Availability is part of the system architecture
8. **The computing environment or architecture** that we can leverage while programming for performance: •Multithreaded programming •Parallel programming
   •Distributed computing •Grid/Cloud computing

9. **Multithreaded programming**

•Modern operating systems work in time-sharing mode. This means the processor availability is frequently switched from virtual processors.

•A thread is a virtual processor that lives within a process

•**Multithreading programming is the ability to program multiple threads together**. (the ability to use multiple processors), often reducing the overall execution time

•Advantages - reducing the overall execution time

•Disadvantages - the predictable number of threads used by the software on a system with an unpredictable number of processor cores available.

10. **Parallel programming**

•Parallel programming adds a dynamic thread number to multithreading programming.

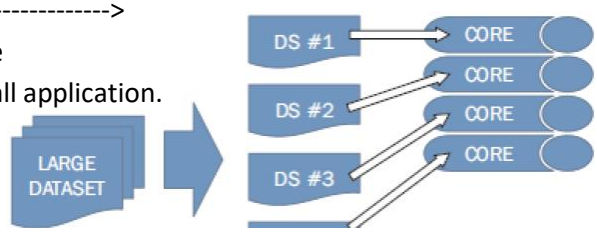
•The thread number is then managed by the parallel framework engine

•Parallelism is the ability to split the computation of a large dataset of items into multiple sub datasets that are to be executed in a parallel way (together) on multiple threads, with a built-in synchronization; the ability to unite all the divided datasets into one of the initial sizes again.

•When using parallel programming, threads flow to the cores trying to use all available resources: ---------->

•The main disadvantage is the percentage of the use of parallelizable code (and data) in the overall application.

• Not everything is parallelizable for system limitations, (hardware resources, external dependencies)




11. **Distributed computing**

•Distributed computing occurs every time we split software architecture into multiple system designs (web services responding on multiple servers with one or more databases )

•The focus is not on speeding up a single elaboration of data, but serving multiple users

•The most popular distributed architecture is the **n-tier**;

•the **3-tier** architecture made by:

- ◦a user-interface layer (any application, including web applications),
- ◦a remotely accessible business logic layer (SOAP/REST web services), and
- ◦a persistence layer (one or multiple databases).

12. **Grid computing**

•Grid computing is a customization of distributed computing, available for huge datasets of highly parallelized computational data. •In grid computing, a huge dataset is divided in tiny datasets.

## 1.2. Architecting High-performance .NET Code

1. **Software architecture**

•application with standard methods, techniques, and tools

•a development team shares them

•Documentation: overall architecture and design, all boundaries that any module must comply with, all technologies to be used by those modules, all standards to apply to, any authentication and authorization logic, the interaction between those modules or layers

2. **Performance concerns about the architecture**

•a single relational DB, an object-relational mapping - good latency and throughput in data extraction from 100 to 10,000 users online (2-tier architectures)

•Splitting the data persistence on different DB breaks constraint-based associations between entities, but gives us great scalability over time (3-tier architectures)

◦slightly worse on latency, scaled to 10,000 or 100,000 online users

3. **Object-oriented design principles**

•**Encapsulation**: Any class can hide its core logic in external items.

•**Inheritance**: Any class can expand the capability of a mother class, by adding more specific properties or adding/changing logics.

•**Polymorphism**: Any object, if extended by inheritance, when compared to other objects of the same parent family, may produce different business results by applying the eventually changed logic as allowed

•**The single responsibility principle** - A single class must have a single responsibility

•**The open-closed principle** -a class must be open for extension and closed for modification

•**The interface segregation principle** - a *role interface* must be created, based on what a client needs with no more logic (methods) or properties than effectively required.

4. **Common designs and architectures**

•A **layer** is a logical module of software with its own core logic and boundaries.

•A **tier** is a physical container of one or more layers, such as a server across a network

•Model-View-Controller, Model-View-ViewModel, The 3-tier architecture, SOA

5. **Common platform architectures**

**5.1 Architecting desktop applications**

•event-driven architecture in Windows Forms •WPF –gives low-latency

• best desktop application relies on a layered MVVM application on the WPF framework

**5.2 Architecting mobile applications**

•similar to desktop applications •usually consumer-oriented applications

•a cloud-based release or web-service based architecture is suggested

**5.3 Architecting web applications**

•load sensitivity (part of scalability) -serving thousands of requests per second, without letting any users feel the traffic on the server

•2-tier architecture or SOA (for multiple web services)

6. **Performance considerations**

•Caching, when and where: reusing a temporary copy of such data for a short time period, reducing the need to contact a persistence storage or any external system, such as a service.

## 2.1. CLR и Асинхронното програмиране

1. **Common Language Runtime** (**CLR**) - the environment which executes any .NET application
2. **CLR ни дава:**

•Memory management •Garbage collection •Working with AppDomains

•Threading •Multithreading synchronization •Exception handling

**2.1 Memory management**

•Memory leak: This occurs anytime we forget to de-allocate memory, or by letting the application always consume more memory;

•Memory corruption: This occurs when we free memory by de-allocating some variable, but somewhere in our code, we still use this memory (because it is referred by another variable as a pointer), unaware of such de-allocation.

**2.2 Garbage collection(GC)**

•GC is the engine that cleans up the memory of managed heap within the CLR with an internal algorithm and its own triggering engine. GC memory cleanup operation is named collect.

**2.3 Working with AppDomains** - an application domain is a kind of virtual application.

•It contains and runs code, starts multiple threads, and links to any needed reference

•Application domains can be created to isolate portions of an application

•Application domains also give us the ability to start multiple applications within a single Windows process

**2.4 Threading**

•A thread is a virtual processor that can run some code from any AppDomain.

•Although at any single time a thread can run code from a single domain, it can cross domain boundaries when needed.

**•thread can be created by starting the Run method of the Thread class.**

•Priority configuration

**•IsBackground property – state for the process**

◦True will signal to the CLR -> non-blocking thread—a background thread (not running state)

◦False (default value) - CLR will consider this thread as a foreground thread (running state)

•The Sleep method suspends the thread for the given time

•Yield method will give the remaining time-slice to the next thread as soon as possible

2.5. **Multithreading synchronization** - Data access in fields and properties must be synchronized!

•CLR guarantees low-level data consistency by always performing a read/write operation

•when multiple threads use multiple variables, it may happen that during the write operation of a thread, another thread could also write the same values, creating an inconsistent state of the whole application

```
public static int simpleValue = 10;
// a static variable with a value per thread instead per the whole process
 [ThreadStatic]
public static int staticValue = 10;
//a thread-instantiated value
public static ThreadLocal<int> threadLocalizedValue = new ThreadLocal<int>(() => 10);
staticValue += 1; simpleValue += 1; threadLocalizedValue.Value += 1;
```


•simpleValue is a simple static integer – unpredictable

•staticValue – duplicates the value for each calling thread

•threadLocalizedValue - The most decoupled value.

3. **Interlocked** - low-level memory-fenced operations such as increment, decrement, and exchange value. All those operations are thread-safe to avoid data inconsistency without using locks or signals. Example: Interlocked.Add(ref value, 4);
4. **Locks -** a lock is a kind of flag that stops the execution of a thread until another one releases the contended resources.

•All locks and other synchronization helpers will prevent threads from working on bad data, while adding some overhead.

•Signaling locks - All those locks that inherit the WaitHandle class are signaling locks. Instead of locking the execution code, they send messages to acknowledge that a resource has become available.

**•In .NET:**

◦the Monitor class - keyword lock { … } allows you to lock access to a portion of code;

◦the Mutex class - Mutex class inherits all features from the Monitor class, adding the ability to synchronize different processes working at the operating-system level;

◦the Semaphore class - lock a specific code portion access.

◦the ManualResetEvent and the AutoResetEvent class - signaling lock classes. Act as flags giving the signal everywhere in our application to indicate whether or not something has happened.

◦Barrier - lets you program a software barrier. It is like a safe point that multiple tasks will use as parking until a single external event is signaled. Example: Barrier (33), folder

**5. Drawbacks of locks - Use lock techniques carefully**

•avoid any race condition - an inconsistent state and/or causes high resource usage

•starvation for resources - a thread never gets access to CPU

•deadlock state - multiple threads wait forever, each with the other, for the same resource or multiple resources without being able to exit this multiple lock state

•Livelock - an infinite loop can occur, wasting CPU time forever

## 2.2. Asynchronous Programming

**1. Understanding asynchronous programming**

•You are cooking in a restaurant. An order comes in for eggs and toast.

◦Synchronous: you cook the eggs, then you cook the toast.

◦Asynchronous, single threaded: you start the eggs cooking and set a timer. You start the toast cooking, and set a timer. While they are both cooking, you clean the kitchen. When the timers go off you take the eggs off the heat and the toast out of the toaster and serve them.

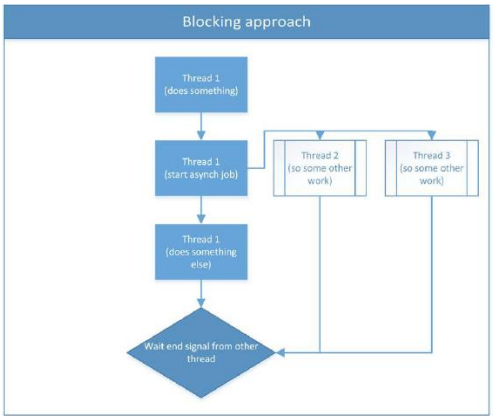
◦Asynchronous, multithreaded: you hire two more cooks, one to cook eggs and one to cook toast. Now you have the problem of coordinating the cooks so that they do not conflict with each other in the kitchen when sharing resources. And you have to pay them.

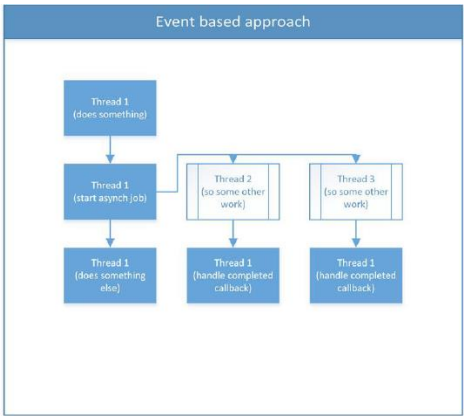
**2. Asynchronous programming theory**

•Two main asynchronous programming designs are available to developers.

◦Blocking one, which happens when the calling thread waits for all asynchronous threads to proceed all together;

◦Event signaling based one, where each asynchronous thread acknowledges the main thread by invoking CLR events



3. **Asynchronous Programming Model (APM)**
   •To start a deferred job, you simply start such a job by using a Delegate (remote method invoker) and then get an object back of type IAsyncResult to know the status of such a remote operation
4. **Event-based Asynchronous Pattern (EAP)**
   •any method that supports synchronous execution will add an overloaded method for an asynchronous invocation. The result will be available only to a specific predefined callback method, one for each available method, within the class itself. Example: event(44), folder
5. **Task-based Asynchronous Pattern(TAP)**
   •**Task – an asynchronous job;**
   •**TAP provides features of APM and EAP with an added signaling lock** like an API that offers a lot of interesting new features:
   ◦Task creation - The Task.Run method immediately returns a Task object that is usable to the query execution status;
   ◦Task synchronization - attached and detached tasks
   - Any task may attach itself to its parent task, if any, although this is not the default behavior
   - With the default behavior, child tasks are detached from their parent tasks. This means that the parent does not care about its child tasks. Examples: (49,50)+folder;

   ◦Task exception handling - Any time an exception happens within a task, any tasks waiting, will receive an AggregateException error that acts as a container for all the exceptions that happened within the tasks being waited on;
   ◦Task cancellation - slight similarity to Thread.Abort method. The difference is that for threads, an exception is raised by CLR itself, immediately stopping the thread's execution;
   ◦Task continuation - the task continuation helps us to select the desired Status property when continuation occurs;
   ◦Task factories - TaskFactory class gives us the ability to start tasks with special options. A TaskFactory class can also be instantiated with custom options that will work as the starting configuration for any task made with this factory.

## 3.1. Async/await

1. **Introduction**
   •Special pattern called **async/await**, which is greatly optimized for cross-thread operations;
   •This pattern helps to achieve asynchronous programming in a **simplified way** transparent ability to execute code on the UI, creating threads without using dispatcher or a delegate.
2. **When to use**
   •I/O-bound needs (such as requesting data from a network or accessing a database);
   •CPU-bound code (such as performing an expensive calculation);
   •Async/await follows the Task-based Asynchronous Pattern (TAP).
3. **Overview of the Asynchronous Model**
   •The core of async programming are the **Task** and **Task<T>**objects, which model asynchronous operations. They are supported by the async and await keywords.
   •For I/O-bound code, you **await** an operation which returns a **Task** or **Task<T>** inside of an **async** method.
   •For CPU-bound code, you **await** an operation which is started on a background thread with the **Task.Run** method.

4. **The await keyword**
   •It yields control to the caller of the method that performed await, and it ultimately allows a UI to be responsive or a service to be elastic.
   •**Examples:**
      ◦ I/O-Bound Example: Downloading data from a web service (You may need to download some data from a web service when a button is pressed, but don't want to block the UI thread.
      ◦ CPU-bound Example: Performing a Calculation for a Game
5. **Key Pieces to Understand**
   •On the C# side of things, the compiler transforms your code into a **state machine** which keeps track of things like yielding execution when an await is reached and resuming execution when a background job has finished;
   •Async code can be used for both I/O-bound and CPU-bound code, but differently;
   •Async code uses **Task<T>**and **Task**, which are used to model work in the background;
   •The **async** keyword turns a method into an async method, which allows you to use the **await** keyword in its body (**await** can only be used inside an async method);
   •When the **await** keyword is applied, it suspends the calling method and yields control back to its caller until the awaited task is complete.
6. **CPU-Bound or I/O-Bound Work?**
   •Will your code be "waiting" for something, such as data from a database? If your answer is "yes", then your work is **I/O-bound**. Use async and await *without* Task.Run;
   •Will your code be performing a very expensive computation? If you answered "yes", then your work is **CPU-bound**. Spawn the work off on another thread *with* Task.Run.
7. **Waiting for Multiple Tasks to Complete**
   •You may find yourself in a situation where you need to retrieve multiple pieces of data concurrently.
   •The Task API contains two methods, **Task.WhenAll** and **Task.WhenAny** which allow you to write asynchronous code which performs a non-blocking wait on multiple background jobs.
8. **Important Info and Advice**
   •async **methods need to have an** await **keyword in their body or they will never yield!**
   •**You should add "Async" as the suffix of every async method name you write (**This is the convention used in .NET to easily differentiate synchronous and asynchronous methods**);**
   •async void **should only be used for event handlers (**async void is the only way to allow asynchronous event handlers to work because events do not have return types);
   •**Write code that awaits Tasks in a non-blocking manner (**Blocking the current thread as a means to wait for a Task to complete can result in deadlocks);

## 3.2. Problems & Solutions

1. **Pausing for a Period of Time**
   •You need to (asynchronously) wait for a period of time. This can be useful when unit testing or implementing retry delays. This solution can also be useful for simple timeouts.
   ◦The Task type has a static method **Delay** that returns a task that completes after the specified time.

- Example 1: defines a task that completes asynchronously, for use with unit testing.

```
static asyncTask<T> DelayResult<T>(T result, TimeSpandelay){
await Task.Delay(delay);
return result;}
```

- Example 2: a retry strategy where you increase the delays between retries.

Exponential backoff is a best practice when working with web services to ensure the server does not get flooded with retries.

```
….      await Task.Delay(nextDelay);
        nextDelay= nextDelay+ nextDelay;….
```

2. **Returning Completed Tasks**

•You need to implement a synchronous method with an asynchronous signature. This situation can arise if you are inheriting from an asynchronous interface or base class but wish to implement it synchronously. You can use Task.FromResult to create and return a new Task<T> that is already completed with the specified value

```
interface IMyAsyncInterface
{
        Task<int> GetValueAsync();
}

class MySynchronousImplementation : IMyAsyncInterface
{
        public Task<int> GetValueAsync()
        {
                return Task.FromResult(13);
        }
}
```

3. **Reporting Progress**

•You need to respond to progress while an asynchronous operation is executing.

◦Use the provided IProgress<T> and Progress<T> types. Your async method should take an IProgress<T> argument; the T is whatever type of progress you need to report.

4. **Waiting for a Set of Tasks to Complete** – Example(folder)

•You have several tasks and need to wait for them all to complete.

◦The framework provides a **Task.WhenAll** method for this purpose. This method takes several tasks and returns a task that completes when all of those tasks have completed:

```
Task task1 = Task.FromResult(3);
Task task2 = Task.FromResult(5);
Task task3 = Task.FromResult(7);
int[] results = await Task.WhenAll(task1, task2, task3);
// "results" contains { 3, 5, 7 }
```

5. **Waiting for Any Task to Complete Example(38,folder)**

•You have several tasks and need to respond to just one of them completing.

◦Use the Task.WhenAny method. This method takes a sequence of tasks and returns a task that completes when any of the tasks complete.

6. **Processing Tasks as They Complete**

•You have a collection of tasks to await, and you want to do some processing on each task after it completes. However, you want to do the processing for each one as soon as it completes, not waiting for any of the other tasks.

◦The easiest solution is to restructure the code by introducing a higher-level async method that handles awaiting the task and processing its result.

7. **Avoiding Context** (the circumstances that form the setting for an event) for Continuations

•When an async method resumes after an await, by default it will resume executing within the same context. This can cause performance problems if that context was a UI context and a large number of async methods are resuming on the UI context.

◦To avoid resuming on a context, await the result of **ConfigureAwait** and pass false for its **continueOnCapturedContext** parameter:

```
async Task ResumeWithoutContextAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);
    // This method discards its context when it resumes.
}
```

8. Handling Exceptions from async Task Methods

•handling exceptions from async Task methods is straightforward.

◦Exceptions can be caught by a simple try/catch, just like you would for synchronous code

```
static async Task ThrowExceptionAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    throw new InvalidOperationException("Test");
}
static async Task TestAsync()
{
    // The exception is thrown by the method and placed on the task.
    Task task = ThrowExceptionAsync();
    try
    {
        // The exception is reraised here, where the task is awaited.
        await task;
    }
    catch (InvalidOperationException)
    {
        // The exception is correctly caught here.
    }
```

## 4.1. PLINQ

1. **Introduction**

•Parallel LINQ (Language Integrated Query) is a parallel implementation of the LINQ pattern;

•PLINQ queries operate on any in-memory IEnumerable or IEnumerable<T> data source, and have deferred execution, which means they do not begin executing until the query is enumerated.

•PLINQ attempts to make full use of all the processors on the system by partitioning the data source into segments, and then executing the query on each segment on separate worker threads in parallel on multiple processors.

2. **The ParallelEnumerableClass**

•includes implementations of all the standard query operators that LINQ to Objects supports, although it does not attempt to parallelize each one;

•<u>contains a set of methods that enable behaviors specific to parallel execution:</u>

◦AsParallel (Example) ◦AsOrdered, AsUnordered ◦WithCancellation ◦ForAll

◦var evenNums= from num in source.AsParallel() where num% 2 == 0 select num;

**2.1 Degree of Parallelism**

•By default, PLINQ uses all of the processors on the host computer.

•With **WithDegreeOfParallelism** method you set the maximum number of processors that will be used in the parallelization

var query = from item in source.AsParallel().**WithDegreeOfParallelism(2) – 2 processors**

**2.2 Ordered Versus Unordered Parallel Queries:** numbers.AsParallel().AsOrdered()

•AsOrdered operator -produces results that preserve the ordering of the source sequence:

◦processed in parallel, but its results are buffered and sorted

◦might be processed more slowly than the default AsUnordered sequence

**2.3 Parallel vs. Sequential Queries**

•Some operations require that the source data be delivered in a sequential manner.

•The ParallelEnumerable query operators revert to sequential mode when it is required.

•For user-defined query operators and user delegates that require sequential execution, PLINQ provides the AsSequential method.

**2.4 Options for Merging Query Results**

•When a PLINQ query executes in parallel, its results from each worker thread must be merged back onto the main thread for consumption by a **foreach** loop, or insertion into a list or array. PLINQ supports the WithMergeOptions method.

**2.5 The ForAll Operator:** query.ForAll(…);

•***foreach*** itself does not run in parallel, and therefore, it requires that the output from all parallel tasks be merged back into the thread on which the loop is running;

•For faster query execution when order preservation is not required and when the processing of the results can itself be parallelized, use the **ForAll** method.

◦ForAll does not perform this final merge step.


**2.6. Cancellation**

•To create a cancelable PLINQ query, use the **WithCancellation** operator on the query and provide a **CancellationToken** instance as the argument.

•When the **IsCancellationRequested** property on the token is set to **true**, PLINQ will notice it, stop processing on all threads, and throw an OperationCanceledException.

**2.7 Exceptions**

•PLINQ uses the AggregateException type to encapsulate all the exceptions that were thrown by a query, and combine those exceptions back to the calling thread.

•On the calling thread, only one try-catch block is required. However, you can iterate through all of the exceptions that are encapsulated in the AggregateException and catch any that you can safely recover from

## 4.2. Synchronization

**1. Introduction**

When one piece of code needs to update----------> **data**

while other code needs to access the same----------^

•**types of synchronization:**

◦*communication* -one piece of code needs to notify another piece of code of some condition (e.g., a new message has arrived)

◦*data protection* -when all three of these conditions are true:

-Multiple pieces of code are running concurrently.

-These pieces are accessing (reading or writing) the same data.

-At least one piece of code is updating (writing) the data.

2. **Example**

•Since this code uses Parallel, we must assume we're running on multiple threads.

```
void IndependentParallelism(IEnumerable<int> values)
{
        Parallel.ForEach(values, item => Trace.WriteLine(item));
}
```

•However, the body of the parallel loop (item => …) only reads from its own data; there's no data sharing between threads here. So, no synchronization of this code is necessary.

3. **Immutable types**

•Immutable types are naturally threadsafe because they *cannot* change; it's not possible to update an immutable collection, so no synchronization is necessary.

•For example, this code does not require synchronization because when each separate thread-pool thread pushes a value onto the stack, it is actually creating a new immutable stack with that value, leaving the original stack unchanged.

•If there wasn't "Peek()" it would need synchronization because each thread pushes a value onto the stack (creating a new immutable stack) and then updates the shared root variable.

```
async Task<bool> PlayWithStackAsync()
{       var stack = ImmutableStack<int>.Empty;
        var task1 = Task.Run(() => Trace.WriteLine(stack.Push(3).Peek()));
        var task2 = Task.Run(() => Trace.WriteLine(stack.Push(5).Peek()));
        await Task.WhenAll(task1, task2);
        return stack.IsEmpty; // Always returns true.}
```

4. **Threadsafe collections**

•Unlike immutable collections, threadsafe collections can be updated. They have all the synchronization they need built in.

5. **Blocking Locks**

•You have some shared data and need to safely read and write it from multiple threads. The best solution for this situation is to use the lock statement. When a thread enters a lock, it will prevent any other threads from entering that lock until the lock is released

•The basic lock statement handles 99% of cases quite well

•There are four important guidelines when using locks:

◦Restrict lock visibility ◦Document what the lock protects. ◦Minimize code under lock.

◦Never execute arbitrary (произволен, случаен) code while holding a lock.

Arbitrary code can include raising events, invoking virtual methods, or invoking delegates.

6. **Locks visibility**

•The object used in the lock statement should be a private field and never should be exposed to any method outside the class •one lock per type

•you should never lock (this) or lock on any instance of Type or string; these locks can cause deadlocks because they are accessible from other code

7. **Blocking Signals**

•You have to send a notification from one thread to another.

•ManualResetEventSlim–most common

◦Any thread may set the event to a signaled state or reset the event to an unsignaled state.

◦A thread may also wait for the event.

•The following two methods are invoked by separate threads; one thread waits for a signal from the other.

```
class MyClass
{
    private readonly ManualResetEventSlim _initialized = new ManualResetEventSlim();
    private int _value;

    public int WaitForInitialization()
    {
        _initialized.Wait();
        return _value;
    }
    public void InitializeFromAnotherThread()
    {
        _value = 13;
        _initialized.Set();
    }
}
```

8. **AsyncSignals -** You need to send a notification from one part of the code to another, and the receiver of the notification must wait for it asynchronously.
9. **Throttling (потискане, дроселиране)**

• *too* concurrent code, and you need some way to throttle the concurrency.

•Code is too concurrent when parts of the application are unable to keep up with other parts, causing data items to build up and consume memory. In this scenario, throttling parts of the code can prevent memory issues.

## 4.3. Scheduling

1. **Scheduler**

•When a piece of code executes, it has to run on some thread somewhere

•A ***scheduler*** is an object that decides where a certain piece of code runs.

•In most cases it is recommended NOT to specify a scheduler whenever possible.

2. **Scheduling Work to the Thread Pool**

•Task.Run is ideal for UI applications, when you have time-consuming work to do that cannot be done on the UI thread.

•don't use Task.Run on ASP.NET unless you are *absolutely* sure you know what you're doing.

• No need to use Task.Run with code executed by Parallel, Parallel LINQ, TPL Dataflow libs

3. **Executing Code with a Task Scheduler**

•Simplest TaskScheduler is TaskScheduler.Default, which queues work to the thread pool.

◦Default for Task.Run, parallel, and dataflow code

• capture a specific *context* and schedule back - TaskScheduler.FromCurrentSynchronizationContext()

4. **ConcurrentExclusiveSchedulerPair -** *two* schedulers that are related to each other

◦The ConcurrentScheduler allows multiple tasks to execute at the same time, as long as no task is executing on the ExclusiveScheduler.

◦The ExclusiveScheduler only executes code one task at a time, and only when there is no task already executing on the ConcurrentScheduler

5. **Scheduling Parallel Code -** You need to control how the individual pieces of code are executed in parallel code.

•Once you create an appropriate TaskScheduler instance, you can include it in the options that you pass to a Parallel method.

6. **Dataflow Synchronization Using Schedulers**

You need to control how the individual pieces of code are executed in dataflow code

◦Once you create an appropriate TaskScheduler instance, you can include it in the options that you pass to a dataflow block.

# Многонишковост в Java

1. **Multithreading** - две или повече задачи, които се изпълняват "видимо" паралелно в рамките на една и съща програма. Въпреки че се нуждае от поддръжката на операционната система, се прилага и изпълнява от самата програма. Необходимо е специално проектиране и планиране на програмата.
2. **Особености**

   - Нишките в Java са обекти от клас Thread.

   - При създаването на нова нишка е необходимо да се предефинира **метода run(),** като цялата функционалност на нишката се вгражда в този метод.

   - **Методът run()** може да вика други методи.

   - Нишката се стартира чрез извикването на **метода start()** на съответния обект.

   - **Методът start()** от своя страна вика **метода run()** и се грижи за изпълняването на всички допълнителни необходими задачи.

   - Нито един от тези методи няма параметри.

   - В една програма могат да се използват много различни Thread класа и много различни обекти от един и същи Thread клас.

   - Всеки от тези класове има свой собствен **run() метод**, който е независим от **run() методите** на останалите класове.
3. **Създаване**

   Класът java. lang. Thread позволява създаването на нови нишки. Всяка нишка трябва задължително да наследи интерфейса Runnable - Изпълняваният код е разположен в нейния метод run()

   Има 2 метода за създаване на Thread :

   **3.1.** производен клас на- java. lang. Thread наследява Runnable –

   - трябва да се предефинира метода run()

   **3.2.** клас наследяващ интерфейса Runnable -- трябва да се дефинира метода run()
4. **Подклас на Thread**

```
class Proc1 extends Thread {
    Proc1() {...} // конструктор ...
    public void run() { .} } ...
 Proc1 p1 = new Proc1(); // създаване нишка p1
p1.start(); // Стартиране на нишката и изпълнение на p1. run()
```

   Пример: First и FirstThread
5. **Наследяване на Runnable**

```
class Proc2 implements Runnable {
        Proc2() { ...} // Конструктор ...
        public void run() { ... // дейност на нишката } }
        ...
        Proc2 p = new Proc2();
        Thread p2 = new Thread( p); ...
        p2.start(); // Стартира нишка, която изпълнява p. run()
```

   Пример:Second

6. **Кой метод да се избере ?**

   6.1. Под-клас на Thread

   - когато класът не наследява вече друг клас (внимание: няма множествено наследяване) - при приложения

   6.2. наследяване на Runnable - когато класът вече е производен - при аплети

   public class MyThreadApplet extends Applet implements Runnable {}

7. **Състояния**

   Състоянието на нишката показва какво в момента тя върши и какво е в състояние да извърши. Тя може да бъде в 4 състояния: нова (New), работеща (Runnable), неработеща, блокирана (Blocked) и завършена (Dead).

8. **Етапи от жизнения цикъл:**
   - **Нова** - нова нишка започва своя жизнен цикъл. Остава в това състояние, докато програмата не стартира нишката.
   - **Runnable** - стартиране на нова нишка - изпълняваща задачата си.
   - **Чакаща** - понякога нишката преминава в чакащо състояние, докато нишката чака друга нишка да изпълни задача. Преминава обратно в работещо състояние само когато друга нишка сигнализира чакащата нишка да продължи да се изпълнява.
   - **Време на изчакване** - текущата нишка може да влезе в състояние на чакане за определен интервал от време. Нишка в това състояние преминава в обратно състояние когато този времеви интервал изтече или когато се случи събитието, което чака.
   - **Прекратена** - текущата нишка влиза в прекратеното състояние, когато завърши задачата си или завърши по друг начин.

9. **Приоритети:**

   JVM управлява приоритетите чрез алгоритъм известен като **fixed priority scheduling**. **Всяка нишка има приоритет.** Приоритетите се представят с цели числа в диапазона от Thread.MAX_PRIORITY (10 - най-висок) до Thread.MIN_PRIORITY (1 - най-нисък). По подразбиране всяка нишка има същия приоритет като тази, която я създала или NORM_PRIORITY (константа 5). След като бъде създадена приоритета на нишката може да се променя с метода setPriority().

10. **Методи**

- **public void start()** - Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
- **public void run() -** If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
- **public final void setName(String name)** - Changes the name of the Thread object. There is also a getName() method for retrieving the name.
- **public final void setPriority(int priority)**
- **public final void setDaemon(boolean on)** - A parameter of true denotes this Thread as a daemon thread.
- **public final void join(long millisec)** - The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
- **public final boolean isAlive()** - Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

**11. Синхронизация между нишки**

- При стартирането на две или повече нишки, те започват да работят асинхронно и независимо една от друга.
- Когато споделят един и същи ресурс е необходимо да не се позволи на една нишка да повреди или смени данните, които в момента се обработват от друга нишка - синхронизация.
- При стартирането на две или повече нишки в една програма, те могат да се окажат в ситуация, при която повече нишки се опитват да направят достъп до едни и същи ресурси и да се получи непредвидим резултат поради конкурентност.
- Например ако повече нишки се опитват да записват в един и същи файл могат да го повредят, защото докато една нишка записва данни, друга може да го затвори.
- Необходимо е действията на множество нишки да бъдат синхронизирани, за да се осигури достъп до определени ресурси само на една нишка. За целта се използва концепцията monitors. Всеки обект се асоциира с монитор, който дадена нишка може да заключи или отключи. Само една нишка в дадено време може да заключи монитор.
- За синхронизация на задачите между нишките се използва блок synchronized. Чрез него се съхраняват споделени ресурси.

```
synchronized(objectidentifier) {
        // Access shared variables and other shared resources
}
```

**12. Контролиране на нишките**

Java предоставя възможности за пълен контрол на нишките в многонишкова програма чрез различни статични методи за контролиране на поведението на нишките.

**- public void suspend()** - This method puts a thread in the suspended state and can be resumed using resume() method.

**- public void stop()** - This method stops a thread completely.

- **public void resume()** - This method resumes a thread, which was suspended using suspend() method.

**- public void wait()** - Causes the current thread to wait until another thread invokes the notify().

**- public void notify()** - Wakes up a single thread that is waiting on this object's monitor.

**13. Комуникация между нишките**

Комуникацията между нишките е важна при приложения, в които две или повече нишки обменят информация помежду си.

- **public void wait() -** Causes the current thread to wait until another thread invokes the notify().
- **public void notify()** - Wakes up a single thread that is waiting on this object's monitor.
- **public void notifyAll()** - Wakes up all the threads that called wait( ) on the same object.

# Нишки. Многонишково програмиране- JavaC

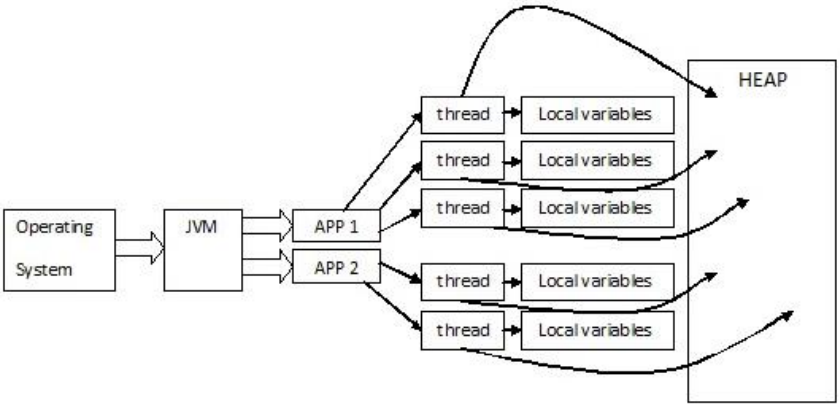
1. **Многозадачност** като понятие в компютърните науки може да бъде разглеждано в два основни аспекта. Базирана на процеси и многозадачност, базирана на нишки.

   1.1 Многозадачност. базирана на процеси :

   - Споделяне на общи хардуерни ресурси – памет, процесорно време.
   - Многопроцесна операционна система.
   - Изпълнение на много процеси едновременно.
   - Процесите са изолирани един от друг по отношение на памет и данни.

   1.2 Многозадачност, базирана на нишки:

   - Споделяне на ресурсите на даден процес.
   - Изпълнение на отделни задачи в рамките на една програма.
   - Нишките са видими в рамките на един процес.
   - Всяка програма има поне една главна нишка, асоциирана с нейния main() метод.
   - Всяка нишка изпълнява последователно даден код от нейната стартова позиция.
   - Всяка нишка изпълнява собствения си код, независимо от всички останали нишки.



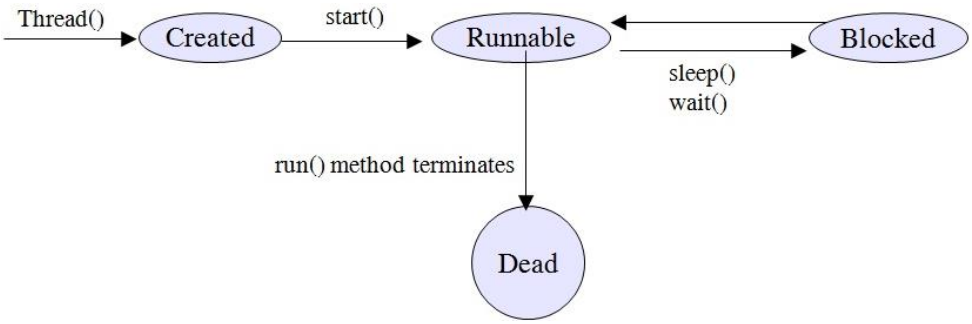

*Изпълнение на паралелни процеси с много нишки*

2. **Предимства на многонишковото програмиране пред еднонишковото:**

   - По – добра използваемост на ресурсите –извършване на няколко изчислителни операции по време на работа с I/O устройства.
   - Опростен програмен дизайн в определени ситуации – разделяне на една програма на независими подпрограми.
   - Разделяне на задачите по приоритет.

3. **Недостатъци**

- Необходимост от допълнителни ресурси(памет и процесорно време) за съхранение на състоянието на дадена нишка при нейното прекъсване и стартиране на нова нишка. Прекъсната нишка, трябва да продължи от последното си състояние
- Допълнителни ресурси в стека за съхранение на локалните променливи на всички методи, имплементирани във всички нишки.




**4. Жизнен цикъл**

- създадена –нишката е създадена, но не е стартирана;
- стартирана – нишката изпълнява дадената задача;
- блокирана – изпълнява блокиращ метод - read();
- завършила -"убита"- е приключила своето изпълнение

**5. Създаване и стартиране на нишки в Java**

В Java съществуват два начина за създаване на нишки:

- Чрез наследяване на класа Thread и предефиниране на метода run(). Методът run() е стартовата точка на всяка една нишка. В това отношение той може да се сравни с main() метода на главната нишка.
- Чрез имплементиране на интерфейса Runnable.

Интерфейсът Runnable представлява празен метод. Неговото съществуване има за цел единствено да ви задължи да предефинирате метода public void run().

*Като заключение може да се каже, че при всички положения инстанцирането на класа Thread е задължително, който от своя страна имплементира Runnable интерфейса.*

**6. Методи**

- **Thread.start()** – стартира дадена нишка;
- **Thread.sleep(милисекунди)** – приспива дадена нишка. Често се използва за приоритет на дадени нишки.
- **Thread.getName()** – връща името на нишката.
- **Thread.Id()** – връща число, с което се асоциира дадената нишка.
- **Thread.getPriority()** – връща приоритета на дадената нишка.
- **Thread.getState()** –връща състоянието на нишката.
- **Thread.interrupt()** – прекъсва дадената нишка.
- **Thread.isAlive()** – връща дали дадената нишка е активна.
- **Thread.join** – чака дадената нишка да приключи.

**7. Синхронизация**

Как да се справим с проблема две нишки да не използват един и същи ресурс едновременно? Отговорът на този въпрос е- чрез използването на синхронизирани методи или синхронизирани блокове код.

```
public void run() {
        synchronized(this){
                .....
        }
}
```

Синхронизацията е важна част от многонишковото програмиране за коректната работа на дадена програма. Важно е да се знае, че от друга страна не трябва да се прекалява със синхронизирането на методи, защото това нарушава принципа на конкурентното програмиране, което е основа на многонишковото програмиране. Добра практика е да не се синхронизират методи, а отделни блокове с код.

## 1.2. Разработване на паралелни програмни приложения

- Новото в паралелизмът е, че е достъпен за всеки компютър и всяка изчислителна платформа.

### 1.2.1. Най-общи форми на паралелизъм

**Конкурентно програмиране**

- диспечерът разпределя процесите – кой процес да използва CPU ресурса в даден момент
- в повечето случаи диспечерът събира процесите толкова бързо, че оставяме с впечатление за псевдо паралелизъм;

**Паралелно програмиране - паралелните системи изпълняват задачите едновременно**

- процесите изпълняват определени задачи едновременно в многоядрена система

**Разпределено програмиране – обменяне на данни чрез съобщения между възлите**

- разпределените системи изпълняват задачи, в рамките на физически разделени възли

### 1.2.2. Шестте предизвикателства

**1) Наследен код (Legacy code)**

**2) Инструментариум (Tools)**

- идеалният сценарий е да има инструменти, които правят целия паралелизъм автоматично
- инструментите трябва да улеснят паралелизма, а не да го затрудняват

**3) Образование (Education)** – наличие на малко специалисти по пар. прогр. във фирмите

- смята се, че клиентите също трябва да бъдат обучени на паралелизъм – целта е да се се определят какви са очакванията от паралелизма.

**4) Страх от многоядрени изчисления (Fear of Many-Core Computing)**

- главно се програмира не 2, 4 или 8 ядра в повечето проекти
- програмирането на 80 или повече ядра изглежда обезсърчително

**5) Възможност за поддръжка (Maintainability)**

**6) Възвръщаемост на инвестициите (Return of Investment)**

кодът ще работи ли по-добре? ; промяната ще увеличи ли продажбите?
няма ли да е по-добре да се закупи по-бърза машина?

### 1.2.3. Създаване на паралелен код – 4 стъпки

**1) Анализ** – откриване на тесни места в приложението, които могат да се паралелизират

- важни въпроси – приложението ми паралелно ли е? ; къде е най-доброто място за паралелизиране? ; как мога да ускоря приложението? ; какво е очакваното ускорение?
- начини за откриване – Intel compiler's loop profiler & profile viewer; Amplifier XE; Advisor XE

**2) Имплементация – добавяне на паралелни конструкции в сорс кода**

- паралелизиране на цикли, части от кода и функции, рекурсивни функции, конвейерни приложения и свързани списъци

**3) Дебъгване (Верификация) – проверка за наличие на грешки с паралелен характер**

- важни въпроси – паралелизмът верен ли е? ; има ли случаи на deadlock или състезания за памет? ; има ли грешки в паметта? приложението ми работи ли както е предвидено?
- използване на Advisor XE; - използване на Inspector XE:

**4) Настройка – настройване на паралелното приложение**

- важни въпроси – задачите в приложението изпълняват ли еднакви количества работа? ; моето приложение мащабируемо ли е? ; threading-а работи ли ефикасно?
- как да настроим паралелния код – тесни места, конкурентност, заключване и чакане

### 1.2.4. Видове паралелизъм

- на ниво инструкции;  - на ниво данни;  - на ниво задача;  - на ниво нишка;  - на ниво памет

### 1.2.5. Ръчна vs Автоматична паралелизация

- Ръчната разработка отнема време, сложен процес, свързана с грешки, итеративен процес.

**Паралелизиращ компилатор (Pre-Processor)** – най-често използваното средство за автоматизирано паралелизиране на последователна програма.

Работи по 2 различни начина:

- изцяло автоматично – компилаторът анализира сорс кода и идентифицира възможностите за парелелизъм. Анализът включва:
  * идентифицира това, което може да блокира паралелизма
  * дали направените разходи за паралелизма реално биха подобрили производителността
  * циклите (do, for) са най-честата цел при автоматичната паралелизация на кода
- направлявано от програмиста – използвайки директиви за компилаторите, програмистът „показва" на компилатора как да паралелизира кода

**Автоматична паралелизация (Caveats)** –могат да се получат грешни резултати

- производителността може да спадне;  - ограничен от особеностите (основно циклите) на кода;
- много по-малко гъвкав от ръчната паралелизация

**Избор на правилна паралелна конструкция** –може да се придържаме към един набор от конструкции или да ги смесваме и съчетаваме с други

**High-Level VS Low-Level Конструкции**

- Високи нива на абстракция
  * няма гаранция, че кодът ще работи паралелно;  това решение се делегира при run time
- Конструкции от ниско ниво – упражняват по-пряк контрол над паралелизма
  * използването на конструкции от най-ниското ниво може понякога да наруши паралелизма и да изисква експертни познания;
  * често са обвързани със специфичен брой ядра и не се мащабират автоматично за многоядрени архитектури

**Паралелизъм на ниво данни VS Общ паралелизъм**

- Паралелизъм на ниво данни - главно се занимава с операции върху масиви от данни
  * някои типове като SIMD се поддържат директно в хардуера на процесора
  * други техники като манипулация на масиви -> чрез библиотеки, разширения на езиците
  * писането на паралелен код на ниво данни води до код, който може да се мащабира и да се възползва от увеличаването на броя ядра
- Общ паралелизъм - изпълнението на отделни задачи едновременно
  * не-цифровият код обикновено се изпълнява от паралелни алгоритми на ниво задача, а не от паралелни алгоритми на ниво данни

## 1.2.6. Паралелизмът и програмистите

**Най-разпространените проблеми при използването на нишки:**
* състезания по данни ; * determinancy races ; *мъртви хватки ; * дисбаланс на товара ;
* Threading/Tasking Overhead ; * Synchronization Overhead ; * грешки по памет
Състезания по данни – как да се реши проблемът?
- Cilk Plus – different kinds of Cilk objects to handle shared data – Reducers, Holders
- OpenMP – different kinds of OpenMP pragmas to handle shared data – Locks, Critical Sections, Atomic Operations, reduction clause
- TBB – concurrent containers, mutexes and atomic operationsA
Мъртви хватки (Deadlocks)
- avoiding by establishing an order in which locks are acquired (a lock hierarchy).
When all threads always acquire locks in the specified order, this deadlock is avoided.

**Poor Load Balancing (дисбаланс на товара)**
- балансиране на товара –действието, което проверява дали всички нишки работят еднакво, използвайки всички налични ядра на процесора
  * добре балансиран работен товар– всички нишки в паралелната програма трябва да извършват еднакви количества работа
  * лошо балансиран работен товар– води до това, че някои нишки са неактивни и това води до пропилян ресурс
Съвети за балансиране на товара
- случай 1: всички задачи са с еднаква дължина
  * разделяне на броя задачи в групи с (почти) еднакъв размер, възложени за всяка нишка
- случай 2: всички дължини на задачите се знаят предварително
  * динамично разделение на задачите, възлагани на нишки, когато дължините на отделните задачи не са еднакви; - статично планиране – ясно упражнение
Динамично планиране
**- Producer/Consumer** - често използван модел, когато има някаква предварителна обработка преди задачите да бъдат налични за Consumer нишките
  * една нишка (Producer) поставя задачите в споделена опашъчна структура, докато Consumer нишките премахват задачите за обработка, както е нужно
**- Boss/Worker**
  * Worker нишките се „срещат” (обменят инфо) с Boss нишката, когато им е нужна повече работа, за да получат възложенията директно
  * трябва да се използва правилния брой и смесица от нишки, за да се гарантира, че нишките, натоварени да изпълняват необх. изчисления, не са неактивни

**Threading/Tasking Overhead**
- пускането на нишки консумира процесорно време, така че е важно на нишките да е възложено голямо количество работа, така че този overhead е незначителен в сравнение с работата, която се изпълнява от нишката
- ако нишката изпълнява малко работа, threading overhead може да доминира върху приложението. Този overhead обикновено е причинен заради твърде фината гранулярност.

**Synchronization Overhead**

- получава се заради използването на твърде много заключвания, бариери, mutexes или други примитиви за синхронизация
- съвети:
  * реструктуираране на кода, така че конструкциите се използват по-малко пъти
  * използване на повече заключвания чрез проектиране на софтуера да използва само атомарни операции
  * използване на инструмент за синхронизация къде се намира този overhead

**Грешки по памет**

- когато последователна програма е превърната в паралелна, всички съществуващи грешки, свързани със заделянето на памет, могат да доведат до това програмата да не може да работи
- паралелното програмиране добавя 2 нови типа грешки по памет
- грешно споделяне
- реално споделяне - вариация на грешното споделяне
  * разликата е, че 2 нишки споделят една и съща променлива
  * 2 ядра, които постоянно четат и записват в същата памет, ще последва в подобно прецакване (повреждане) на паметта и ще засегне производителността

## 1.3. Синтез на паралелни алгоритмични решения

### 1.3.1. Основни аспекти на алгоритмичния анализ

- от теоретична гледна точка алгоритмите могат да бъдат трансформирани от последователна в паралелна форма или от една паралелна форма в друга

1) Паралелизация на съществуващите последователни алгоритми или модифициране на техни части, в които има заложен потенциален паралелизъм

2) Синтез на нов паралелен алгоритъм от съществуващ паралелен алгоритъм

**1. Разглеждат се** машинно независимите аспекти като степен на паралелизъм и възможности за декомпозиция.

**2. Разглеждат се** машинно зависимите аспекти като брой на процесорите и допълнителни разходи за комуникация и координация на паралелните процеси.

**Важни въпроси** – коя архитектура е най подходяща за решаването на даден проблем? ; каква ефективност можем да получим при използването на дадена комп архитектура?

Корелация между алгоритмично и архитектурно пространство

| **Паралелен алгоритъм** | | **Паралелна архитектура** |
|---|---|---|
| гранулярност | --> | сложност |
| ниво на паралелизъм | --> | режим на работа (обработка) |
| структури от данни | --> | организация на паметта |
| communication media | --> | system area network |
| размер на алгоритъма | --> | брой процеси |
| подход за програмиране | --> | категория компютърна архитектура |

### 1.3.2. Стилове за паралелно програмиране

- съвременните стилове са от типа MIMD
- в зависимост от оползотворяваното ниво на паралелизъм
  * паралелизъм по данни – SPMD (Single Program Multiple Data)
  * функционален паралелизъм – MPMD (Multiple Programs Multiple Data)
- в рамките на една и съща паралелна програма могат да се комбинират и двата стила на паралелно програмиране

### 1.3.3. Парадигми за синтез на паралелни алгоритми

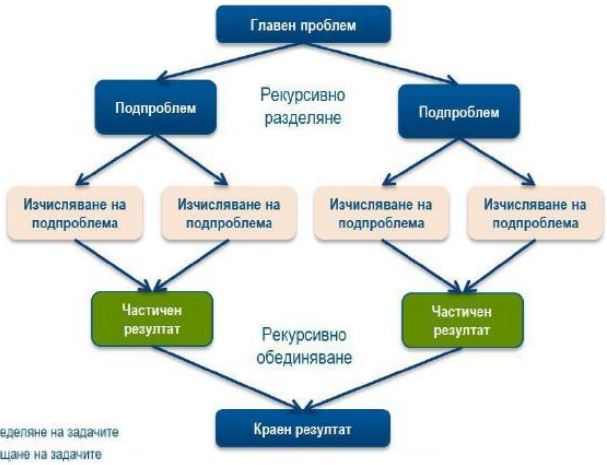
- дефинират начините за структуриране на паралелните алгоритми за изпълнението им от паралелна компютърна система
- всяка парадигма предстява клас алгоритми
- в рамките на една и съща паралелна програма няколко парадигми могат да бъдат комбинирани по различни начини

### 1.3.4. Паралелни програмни модели

- Една програма – множество данни (SPMD)
- Разделяй и владей (Divide – Conquer)
- Главен-подчинени (Master/Slave, Manager/Workers, Task-farming)
- Конвейеризация на паралелните изчисления (Pipelining)
- Работен пул (Work pool)
- Фазово –паралелен (Phase parallel)
- Синхронни и асинхронни итерации (Synchronouse and asynchronous iterations)

**Една програма – множество данни (SPMD)**


**Разделяй и владей**




**Главен-подчинени**


- главният процес изпълнява основната последователна част на програмата и разпространява паралелния работен товар към множество подчинени процеси
- когато някой от подчинените процеси завърши изпълнението на текущата задача, той информира главния процес, който от своя страна му задава за изпълнение нова задача

**Конвейеризация на паралелните изчисления**


- прилага се в случаите, при които изпълнението на дадена последователност от операции може да се припокрие; **-** множество процеси формират виртуален конвейер;
- потокът от данни се подава на конвейера, като в различните фази на конвейера процесите се изпълняват едновременно, т.е. осъществява се припокриване на едновременната обработка на различни фази на множество процеси; **-** типът на паралелизма е по данни

**Работен пул (Work pool)**


- създават се множество паралелни процеси
- всеки свободен процес може да вземе произволен работен товар от пула и да го изпълни, като в резултат на това изпълнение може да генерира допълнителен изчислителен товар и да го добави в работния пул
- изпълнението на паралелната програма завършва, когато се изчерпи изчислителният товар в работния пул

**Фазово –паралелен** - изчислението включва определен брой суперстъпки, всяка включваща 2 фази:

* Изчислителна фаза – множество процеси изпълняват независими изчисления

* Координираща фаза – процесите взаимодействат чрез синхронни операции като например бариерна синхронизация или блокираща комуникация

**Синхронни и асинхронни итерации**

- специален случай на фазовата парадигма, където суперстъпките представляват последователност от итерации в рамките на цикъл


* Синхронни итерации – никой от паралелните процеси не може да започне (i+1) –тата итерация на цикъла преди всички процеси да са завършили i-тата
* Координираща фаза – процесът продължава изпълнението на следващата итерация, без да изчаква останалите процеси да завършват изпълнението на текущата итерация

## 1.3.5. Етапи при създаването на паралелна програма

**Намиране на конкурентност**

- Идентифициране и анализиране на използваема конкурентност

**Структура на алгоритъма**

- Структуриране на алгоритъма, за да се възползваме от потенциална конкурентност

**Поддържащи структури**

- Дефиниране на програмни структури и структури от данни, необходими за кода

**Механизми за имплементация**

- Нишки, процеси, съобщения и т.н.

Пример: Паралелизиране на Bubble Sort

**Намиране на конкурентност** - подходът сравни-размени се нуждае от подредба на операциите

**Структура на алгоритъма**

- Идеята на алгоритъма предполага, че данните са зависими
- Идея: паралелизиране на отделните последователни операции (сравни-размени?)
- Сортиране за нечетни-четни – Сравняване на [нечетен][четен] – индексирани двойки и размяна при случай

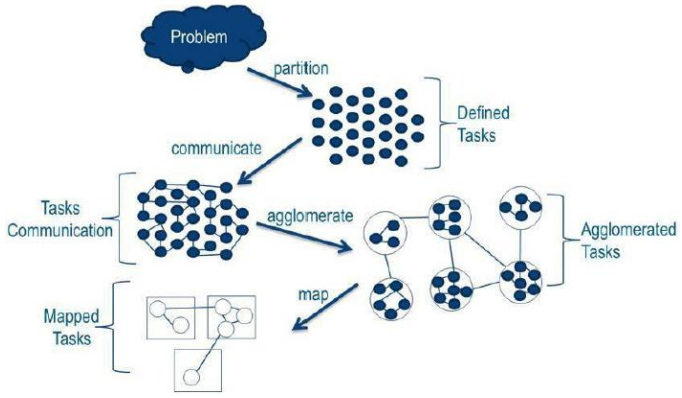
**Поддържащи структури** - Споделени данни

**Механизми за имплементация** – нишки

## 1.3.6. Синтез на паралелни алгоритми (Методология за синтез на Ян Фостер)

**Методология за синтез PCAM**

**Partitioning** – разделяне на части; **Communication** – Комуникация;
**Agglomeration** – натрупване в групи (купове); **Mapping** – Мапване



**Фаза “Разделяне” (Partitioning)**

- дефинират се максимален брой малки паралелни задачи с цел да се постигне декомпозиция на фини гранули; - броят на процесорите се игнорира, цели се откриване на възможности за паралелно изпълнение

Декомпозиция на областта

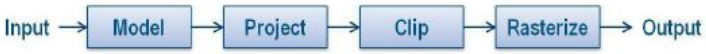
1) Анализ на данните, асоциирани с проблема
   * входни/изходни данни за програмата, междинни резултати
2) Подходящо разделяне
   * възможни са различни размери в зависимост от структурата на данните
   * по възможност частите да бъдат с приблизително еднакъв размер
3) Асоцииране на дадено изчисление с данни
   * асоцииране на всяка операция с данните, над които се прилага

Функционална декомпозиция

1) Декомпозиция на изчисленията на отделни задачи
2) Декомпозиция на данните
   * изцяло независими данни -> пълна декомпозиция
   * висока степен на припокриваемост

**Конвейеризация на изчисленията (Pipelining)** - специален тип функционална декомпозиция

- обработка на един набор от данни в 4 стъпки:

- пример с 3D рендиране: 


- 2 набора се обработват в 5 стъпки (по диагонал надясно)
- 5 набора от данни се обработват в 8 стъпки

**Фаза “Комуникация”**

- изграждане на адекватна комуникационна структура на приложението
- създаване на алгоритми за координация на паралелното изпълнение на задачи
- анализира възможността за възникване на конфликти

- 2 комуникационни етапа в един алгоритъм:

1. Дефиниране структурата на комуникационните канали
2. Определяне на съобщенията, изпращани и получавани чрез комуникационните канали.

Комуникационни шаблони

**Local/Global**

- Local: всяка задача комуникира с ограничено множество от другите задачи (“съседи”)
- Global: всяка задача комуникира с множество задачи

**Structured/Unstructured**

- Structured: задачата и нейните съседни задачи формират регулярна структура – дърво/грид
- Unstructured: комуникационната структура може да бъде произволен граф

**Static/Dynamic**

- Static: идентичността на комуникационните партньори не се променя с течение на времето
- Dynamic: идентичността на комуникационните партньори може да зависи от данните, изчислявани по време на изпълнението, и може да бъде силно променлива

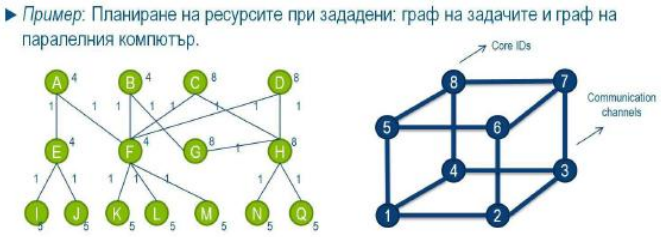
**Synchronous/Asynchronous**

- Synchronous: двойката Producers/Consumers работят координирано, като си сътрудничат при операциите по предаване на данни
- Asynchronous: комуникацията може да изисква потребителят (Consumer) да получи данни без съдействието на производителя (Producer)

**Фаза “Агломерация”**

- преход от абстрактния към конкретния алгоритмичен синтез
- задачите се групират, като се вземат предвид особеностите на имплементацията и факторите, определящи производителността
- ревизиране на получените резултати в първите 2 фази с оглед на максимално адаптиране на структурата на алгоритъма към особеностите на целевата компютърна архитектура

**Фаза “Планиране”**





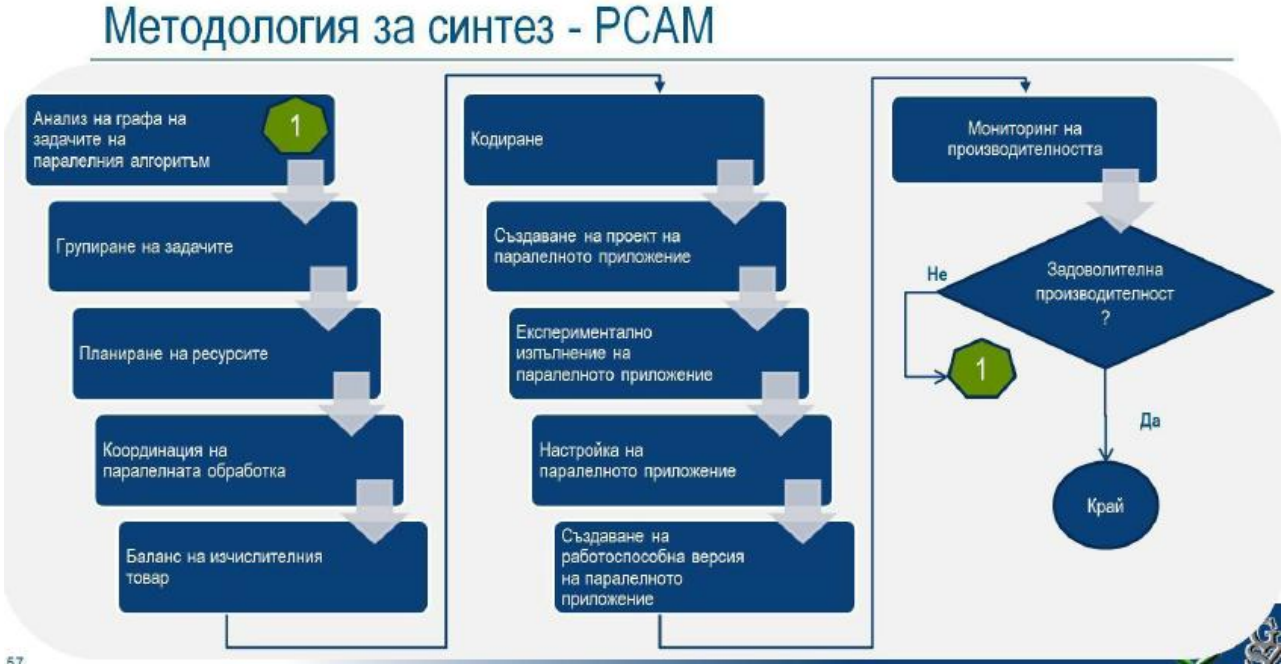
Методология за синтез – PCAM

- след преминаване на всички етапи на проектиране и имплементиране на паралелно приложение, се оценява неговата ефективност
- според резултатите от анализа може да се преразгледат и направят промени по някои (или всички) фази -например промяна в разделянето на подзадачи

Качества на алгоритмичния синтез

Оценява се като се анализират следните му атрибути:

- Паралелизъм на ниво данни (SIMD); - Функционален паралелизъм (MIMD)
- Гранулярност на процесите; - Гранулярност на данните
- Степен на паралелизъм; - Еднородност на операциите
- Синхронизация; - Зависимост по данни; - Генериране и терминиране на процесите



## 1.4. Синтез на паралелни алгоритмични решения – Case Studies

### 1.4.1. Паралелизация на матрични изчисления


**Етапи:**

**1) Определяне на подзадачите**

- Организация на паралелните изчисления
- Разделяне на задачи на подзадачи

**2) Определяне на информационните зависимости:**

* Какво е необходимо за изчислението на един ред от резултантната матрица?

* Какво да се съдържа във всяка подзадача -> редове от матрицата A - колко?;

-> стълбове от матрица B - колко?

* Каква да бъде общата схема на предаване на данните съобразно избрания подход за декомпозиция на задачата на подзадачи?

**3) Мащабиране и разпределяне на подзадачите по процесори**

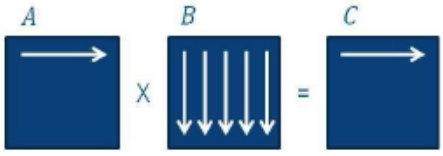
- изчислителна трудоемкост за определените подзадачи; обем на предаваните данни
- Какъв метод за разпределение на базовите задачи по процесори да бъде приложен?

**Case Study 1: Паралелизация на матрични умножения**

▶ Последователен алгоритъм за умножение на матрици

```
void matrix_multiply (int m, int n, int p,
    double **A, double **B, double **C)
    {
       int i, j, k;

       for (i = 0; i < m; i++)
          for (j = 0; j < n; j++) {
             double tmp = 0.0;
             for (k = 0; k < p; k++)
               tmp += A[i][k] * B[k][j];
             C[i][j] = tmp;
          }
       return;
    }
```




$C_{m,n} = A_{m,p} \times B_{p,n}$

Сложност на алгоритъма: $O(m * n * p)$

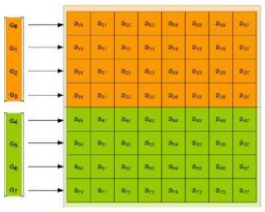
**Метод 1: Лентово разделяне (Block-Striped Decomposition)**

- Дребнозърнест подход - основната подзадача е изчислението на един елемент на резултантната матрица C. Всяка подзадача трябва да съдържа по един ред на матрицата A и един стълб на матрицата B.
- Общият брой основни подзадачи е $n^2$ (броя на елементите на матрица C). Достигнатото ниво на паралелизъм е в повече, излишно!
- По правило броят налични процесори е по-малък от $n^2$ (p< $n^2$)

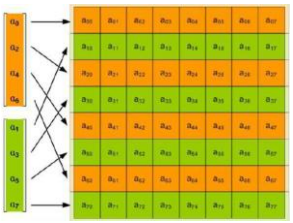
Обобщена подзадача

- определяне в една подзадача не само на един, а на няколко елемента на резултантната матрица; - при определяне като базова задача да се изчислява по един ред от резултатната матрица C, като общият брой на подзадачите става n.

**1) Хоризонтално разделяне на данните, по редове**



**Редуване(цикличност) на редовете или стълбовете**



**2) Анализ на информационните зависимости XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX**

**3) Мащабиране и разпределяне на подзадачите по процесори**

1. Създаване на обобщени подзадачи

- изчисленията могат да бъдат обобщени по такъв начин, че всеки процесор ще изпълни няколко скаларни умножения на редовете на матрица A и колони на матрица B
В този случай, след приключване на изчисленията, всяка обобщена основна подзадача ще е изчислила няколко реда на резултатната матрица C

2. Декомпозиция

- първоначалната матрица A се декомпозира на p хоризонтални ленти, а матрица B на p вертикални ленти

3. Разпределяне на задачите по процесори

- разпределяне на подзадачите между процесорите
- разпределението трябва да отговаря на изискванията за ефективно представяне на ринговата структура на информационните зависимости

**Анализ на ефективността**

1. Оценка на трудоемкостта на алгоритъма

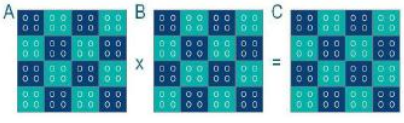
- Оценява се броят изчислителни операции, необходими за решаването на поставената задача, без да се отчита времето за предаване на данните между процесорите, а продължителността на всички изчислителни операции се приема за една и съща;
- Важна е преди всичко сложността на алгоритъма, а не точното време за изпълнение на изчисленията.

2. Теоретични оценки (предсказване на времето за неговото изпълнение)

- Формират се много по-точни съотношения с отчитане на продължителността на изпълнение на операциите и трудоемкостта на комуникационните операции.
- Точността на получените изрази се проверява посредством изчислителни експерименти, по резултатите от които времето за изпълнение се сравнява с теоретично определеното.

**Метод 2: Блоково разделяне (Chessboard Block)**

- разпределяне по данни – по схема тип „шахматна дъска“




- основна подзадача – процедура за изчисление на всички елементи на един блок на матрица C

**Последователна имплементация:** разделяне матриците A и B на 4 подматрици. Всяка матрица се разглежда като един елемент и се умножава; 8 умножения на подматрици, 4 сумирания

**Паралелна имплементация**

- основна подзадача – процедурата за изчисление на всички елементи на един блок на матрица C. За всеки процесор трябва да бъдат достъпни съответният набор от редове за матрица A и стълбове на матрица B.
- разполагането на необходимите данни за всяка задача ще доведе до нарастване на обема на използваната памет
- изчисленията трябва да бъдат организирани по такъв начин, че във всеки момент от време подзадачите да съдържат само част от необходимите за изчисленията данни, а достъпът до останалата част от даннаите да се осъществява на база предаване на данни между процесорите

**2 алгоритъма за паралелна имплементация– на Fox, на Cannon**

**Алгоритъм на Fox**

Анализ на информационните зависимости

- базовите подзадачи отговарят за изчисление на отделните блокове на матрица C
- на всяка итерация в подзадача се планира да бъде разположен само един блок от изходната матрица A и един от изходната матрица B
- наборът от подзадачи образува квадратна решетка, съответстваща на структурата на блоково представнета матрица C
- подзадача номер (i,j) отговаря за изчислението на блок Cij на резултатната матрица C.

В резултат подзадачите формират двумерна решетка q x q.

**1) Етап на начална инициализация** – на всяка подзадача (i, j) се предават блокове A(i,j)a, B(i,j) и се нулират блокове C(i,j)

**2) Етап на изпълнение**

Мащабиране и разпределяне на подзадачите по процесори

- размерите на матричните блокове могат да бъдат избрани така, че броят на подзадачите да съвпада с броя на наличните процесори p
- най-ефективно паралелно изпълнение на този алгоритъм може да бъде постигнато при топология на комуникационна мрежа тип двумерна решетка
- в такъв случай разпределянето на подзадачите между процесорите се извършва по естествен начин: подзадача (i,j) трябва да бъде разпределена на процесор Pij

**Алгоритъм на Cannon**

Основна разлика с алгоритъма на Fox

- умножението на блоковете в рамките на подзадача да стане без допълнителен трансфер на данни; - преместването на блокове между подзадачите в хода на изчислителния процес да се осъществява с помощта на прости комуникационни операции

Анализ на информационните зависимости

**1) Етап на начална инициализация:**

- Стартира се с такова разпределение. че първото умножение между блокове да се извърши

без допълнителен трансфер на данни:
- На всяка подзадача (I,j) се предават блокове A(i,j), B(i,j) и се нулират блокове
- За всеки ред i от решетката от подзадачи блоковете на матрица A се прехвърлят на (i - 1) наляво;
- За всеки стълб от решетката от подзадачи на матрица B се прехвърлят на j - 1 позиция нагоре,

**2) Етап на изпълнение:**

- Преразпределение на матричните блокове на първа стъпка на алгоритъма.
- След разпределението. по време на първата стъпка. матричните блокове могат да бъдат умножени без допълнителни операции по препредаване на данни.
- За получаване на всички останали блокове след операцията "умножение на блокове" се извършватследните операции:
  * Блоковете на матрица A се преместват с една позиция наляво по протежение на реда;
  * Блоковете на матрица B се преместват с една позиция нагоре по протежение на колоната

Мащабиране и разпределяне на подзадачите по процесори

- Размерите на матричните блокове могат да бъдат избрани така, че броят на подзадачите да съвпада с броя на наличните процесори p
- Най-ефективно паралелно изпълнение на този алгоритъм може да бъде постигнато при топология на комуникационна мрежа тип двумерна решетка.
- В такъв случай разпределянето на подзадачите между процесорите се извършва по естествен начин: подзадача (i, j) трябва да бъде разпределена на процесор Pij.

Имплементация

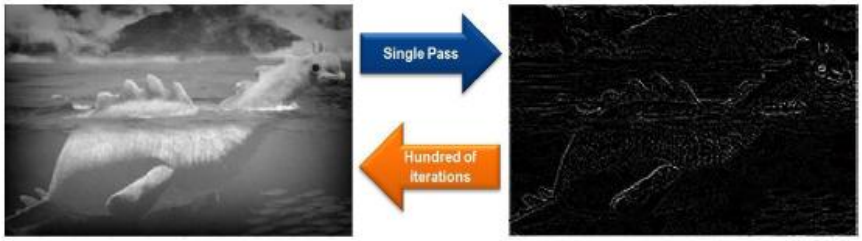
Целева платформа – клъстер от работни станции; **Програмен модел – Главен – подчинен**; Имплементация – MPI; Изискване на алгоритъма – броят процесори да бъде точен квадрат; процесите се организират в комуникатор с виртуална декартова топология

**Паралелизация на матрични изчисления: Изводи**

Три паралелни метода за матрично умножение

- Алгоритми, основани на лентова декомпозиция на матриците между процесорите в два варианта:
  * Хоризонтална декомпозиция на матрица A и вертикална декомпозиция на матрица B.
  * Хоризонтална декомпозиция за A и B.
- Алгоритми на основата на блокова декомпозиция
  * Алгоритъм на Fox и алгоритъм на Cannon
    -> Основават се на използването на една и съща схема за поблокова декомпозиция.
    -> Отличават се по начина на изпълнение на операциите по предаване на данни.
    -> Алгоритъм на Fox: трансфер и циклично препредаване на блокове
    -> Алгоритъм на Cannon: циклично препредаване на блокове
- Различните начини за декомпозиране на данните водят до различии топологии на комуникационната мрежа, при конто изпълнението на паралелните алгоритми е най-ефективно.
  * За реализация на алгоритмите на основата на лентова декомпозиция най-подходящите топологии на комуникационната мрежа са хиперкуб или пълно свързана.
  * За реализация на алгоритмите, основани на блокова декомпозиция, е необходимо наличие на топология решетка.

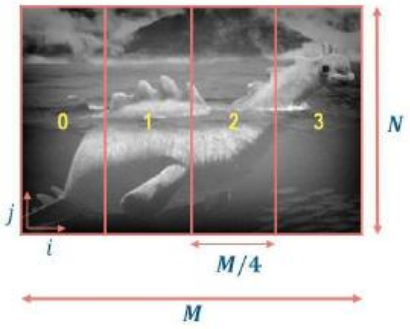
### 1.4.2 Case Study 2: Откриване на ръбове (Edge Detection)



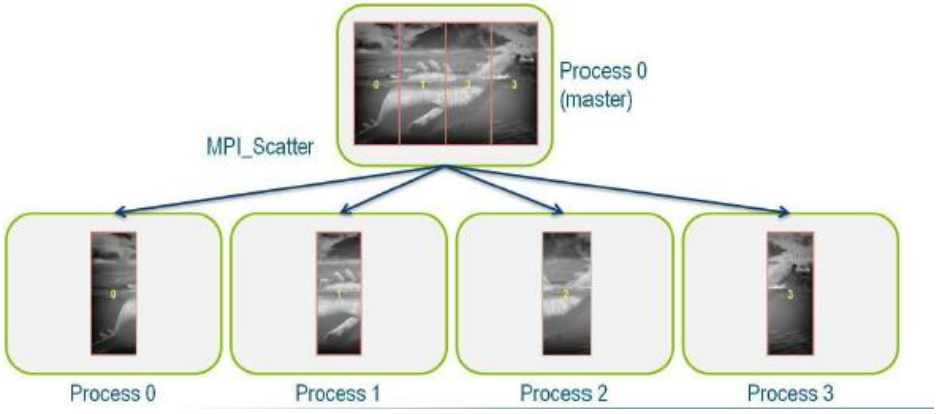

- Сравняване на всеки пиксел с неговите четири най-близки съседи.
- Стойности на пикселите: от 0 (черно) до 255 (бяло)

$$edge_{i,j} = image_{i+1,j} + image_{i,j+1} + image_{i-1,j} + image_{i,j-1} - 4image_{i,j}$$

**Декомпозиционен подход при 4 процеса**



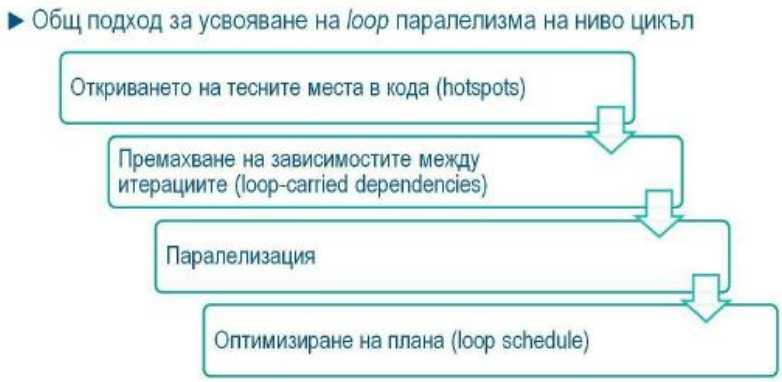

- The slices are numbered according to the rank of the process that owns them.
- It assumes that $M$ and $N$ are exactly divisible by the number of processes $P$.
- The entire file is read into an array on a master process (rank = 0) and then distributed amongst the other processes $P$.
  - ! Note that this is not particularly efficient in terms of memory usage as we need enough space to store the whole image on a single process.



## 1.5. Оптимизации на паралелна производителност

### 1.5.1. Оползотворяване на LLP (Loop-Level Parallelism) паралелизма

- Почти всички HPC кодове използват итеративни конструкции, т.е. цикли;
- OpenMP се фокусира върху паралелизацията на цикли - особено полезно при паралелизация на съществуващ код - Основно преструктуриране на кода е непрактично / ненужно.
- Цел на оползотворяването на LLP - трансформиране на последователния код в паралелен
  * Чрез трансформации, конто не повлияват програмната семантика
- LLP работи добре за системи, базирани на архитектурния модел с обща памет
  * Ограничена мащабируемост извън този вид системи.




**Паралелизация на цикли -** Обособяване на независими подзадачи за паралелно изпълнение

Ограничения:

- Правилната работа на програма не би трябвало да зависи от това коя точно нишка изпълнява итерацията на паралелния цикъл;
- Не се допуска използване на предварителен изход (преди края на цикъла) от паралелен цикъл.

**Паралелизация на цикли с OpenMP**

```
#pragma omp parallel for
```

**Опции:**

private(списък); firstprivate(списък); lastprivate(списък); reduction(оператор:списък); schedule(type[, chunk]); collapse(n); ordered; nowait;

**Примери за останалите опции --> лекция 1.5 от стр. 5 до стр. 12**

**Опция ordered** – указва, че в цикъла може да се очаква директива `#pragma omp ordered`, която дефинира блок в тялото на цикъла, чието изпълнение трябва да следва реда на итерациите

**Опция collapse** – указва, че *n* последователно вложени цикъла се асоциират с тази директива. В този случай се образува общо пространство от итерации, които се разпределят между нишките в паралелния участък

**Планиране на итерациите за паралелизируеми цикли (Loop Scheduling)**

- производителността зависи от системните разходи, произтичащи от 4 основни фактора:
  * дисбаланс на работното натоварване (load imbalances) между процесорите
  * натоварване от синхронизация (synchronization overhead)
  * комуникационно натоварване (communication overhead)
  * координация на нишки (thread management overhead)

**Пример:** Инициализация на масив `for (i=0; i<N; i++) a[i] = 0;`


Опция **schedule(type[, chunk])** – указва по какъв начин да бъдат разпределени итерациите между нишките. Параметър **<type>** задава следните основни разпределения:

* **статично (static)** – указва блоково-циклично разпределение на итерациите

* **динамично (dynamic)** – указва динамично разпределение на итерациите с фиксиран размер на блока. За начало всяка нишка получава по n (chunk size) итерации. След обработването им нишката получава първата свободна итерация от n-те. Освободилите се нишки получават нови порции итерации и така до тяхното изчерпване.

* **направлявано (guided)** – указва динамично разпределение на итерациите, но с намаляващ (до стойността на chunk) размер на блока, пропорционално на броя на все още неразпределените итерации, разделено на броя нишки, изпълняващи цикъла.

**`#pragma omp for schedule(auto)`** – Методът на разпределение на итерациите се избира от компилатора или от системата за изпълнение. Параметър chunk не се задава.

**`#pragma omp for schedule(runtime)`** – Методът на разпределение на итерациите се избира по време на изпълнение на програмата съобразно стойността на променлива на средата OMP_SCHEDULE.

- изборът на разпределение и стойност за параметър chunk зависят от характера на изчисленията вътре в цикъла

**- Целта е: оптимално разпределение на изчислителния товар между нишките.**
**В този случай се достига максимална ефективност.**

Пример: Намиране на прости числа в определен интервал, паралелизация и оптимизации:
**Виж лекция 1.5 -> стр. 13,14,15 или https://imgur.com/a/aJECe**

**Трансформация над цикли**

**Основна цел:** организиране достъпа до данни по такъв начин, че често използваните данни да бъдат на разположение в кеш паметта

▷ **Пример за 'добър' достъп до паметта** – Масивът се достъпва по редове.

```
1  for (i=0; i<n i++)
2      for (j=0; j<n; j++)
3          sum += a[i][j];
```

▷ **Пример за 'лош' достъп до паметта** – Масивът се достъпва по колони. Лошо оползотворяване на паметта. Производителността се влошава с увеличаване на размера на обработвания масив

```
1  for (j=0; j<n j++)
2      for (i=0; i<n; i++)
3          sum += a[i][j];
```

Вложените for цикли могат да имат зависимости по данни, които пречат на паралелизацията.

Стратегии:

- Разделяне на цикъл (Loop fission)

При наличие на циклична зависимост между итерациите

* Разделяне на цикъла на два или повече цикли
* Паралелна обработка на новополучените цикли

- Обединяване на цикли (Loop fusion)
* Обратно на loop fission
* Комбиниране на цикли за увеличаване на размера на гранулата

- Инвертиране на цикъл (Loop inversion)

Инвертирането на вложените for цикли способства за:
* разкриване на паралелизируем цикъл
* увеличаване на размера на гранулата
* подобряване на програмната локалност (program's locality)

- Развиване на цикъл (Loop unrolling)

### 1.5.2. Векторизация на кода

Паралелизмът по данни чрез векторни инструкции допълва многоядрения паралелизъм. Векторните инструкции са една от имплементациите на SIMD паралелизма.


**Начини за векторизиране на кода**

- Използване на възможно най-простите начини (с ниска трудоемкост).
- Максимално оползотворяване възможностите на компилатора за създаване на векторен код.

Това гарантира, че с появата на ново поколение процесори и нови системи от инструкции няма да се налага пренаписване на кода. Ще бъдат необходими само нов компилатор и прекомпилация на вече съществуващия код.

**Векторни инструкции – концепция (псевдо код)**

- Векторният цикъл (вдясно) изпълнява 1/4 от броя на итерациите на скаларния цикъл (вляво) и всеки оператор за сумиране действа едновременно върху 4 елемента в даден момент (т.е. сумирането тук е единична инструкция върху множество данни).

| Scalar Loop | Vector Loop |
|---|---|
| for (i = 0; i < n; i++) | for (i = 0; i < n; i += 4) |
| A[i] += B[i]; | A[i:(i+4)] += B[i:(i+4)]; |

* **Максималното потенциално ускорение** на това векторизирано изчисление по отношение на скаларната версия е равно на броя стойности, едновременно съхранявани във векторни регистри на процесора. В горния пример това ускорение е равно на 4.

Практическото ускоряване при векторизация зависи от дължината на векторните регистри, типа на скаларните операнди, вида на инструкцията и комуникацията с паметта.

- Общата особеност на всички набори от инструкции е, че те поддържат паралелизъм на ниво данни, т.е. прилагане на една и съща аритметична операция към набор от данни (към вектор).
- Броят на елементите във всеки вектор е отношението на дължината на векторния регистър към размера на типа данни.
- Тъй като наборите от векторни инструкции се развиват, те поддържат по-дълги вектори, повече типове данни и по-голямо разнообразие от инструкции.

**Процесорни архитектури Intel Xeon и MIC**

- Всяко ядро на процесор Intel Xeon или Intel Xeon Phi копроцесора има свои собствени средства за векторна обработка и следователно различии ядра могат да работят с различии векторни инструкции във всеки отделен момент (т.е. ядрата не работят в " lock-step“ режим).
- Програмистът „споделя" отговорността с компилатора, за да се гарантира, че там, където приложението може да се възползва от паралелизма на данните, да се използват векторни инструкции.
- Векторизация може да съществува и във всяка нишка на многонишкова програма, което допълнително усложнява работата на програмистите.

**Използване на векторни инструкции: 2 подхода автоматична/явна векторизация**

**Начини за векторизиране на кода – от най-лесно ----> към най-трудно**

- Автовекторизация (без промени в кода)
- Директиви за векторизация (pragma simd – Intel Cilk Plus; pragma omp simd – OpenMP 4.0)
- Intel Cilk Plus Array Notation
- Класове SIMD intrinsic – F32vec, F64vec
- Векторни intrinsic функции - __mm_fmadd_pd(…); __mm_add_ps(…)a
- Асемлбер

<u>- с използване на директиви на компилатора (примерни за Intel):</u>

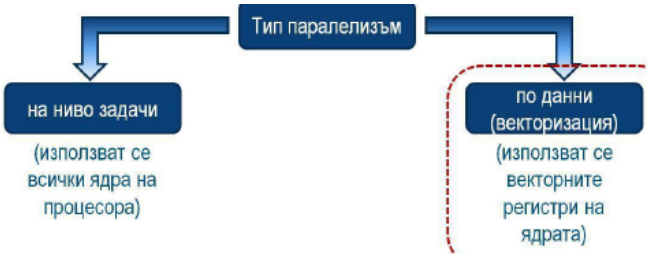
**#pragma ivdep** – указва на компилатора, че в цикъла няма зависимости между итерациите
**#pragma vector always** – указва на компилатора да не се съобразява с политиката за ефективност при използването на векторизация

<u>с използване на инструментите на Intel Cilk Plus – директива</u>

**#pragma simd** – забранява всички проверки от страна на компилатора относно възможностите да векторизира кода. На практика се предоставя на разработчика да извърши това. анни, а е зададено да се направи векторизация, то програмата няма да работи коректно.
**Intel Cilk Plus Array Notation** – прост синтаксис, наподобяващ Fortran:

<u>- OpenMP v4.0</u>

* всяка нова версия на OpenMP е свързана с усъвършенстване на стари и добавяне на нови директиви, улесняващи паралелизацията
* разширяване на поддържаните типове паралелизъм за системи с обща памет - добавяне на директиви за работа с паралелизъм по данни




**#pragma omp simd [clause[ [,] clause]…]** – указва на компилатора да векторизира цикъла. Векторизацията с тази прагма допълва възможността за напълна автоматична векторизация от страна на компилатора

**#pragma omp parallel for simd [clause [ [,] clause]…]** – указва създаване на паралелна област, разпаралелване на итерациите на цикъла и прилагане на векторизация

**Автоматична векторизация (auto-vectorizer)**

- Компонент на Intel компилаторите;
- Компилаторът самостоятелно идентифицира подходящи за векторизация цикли в изходния код и генерира за тяхното изпълнение пакет от SIMD инструкции;
- Автоматичната векторизация се поддържа в IA-32 и Intel® 64 архитектурите.
- Осъществява се само за цикли, отговарящи на определени условия.
- Възпрепятстващи фактори: Обръщения към функции (Function calls)
  * Входно/изходни операции (I/O operations)
  * Осъществяване на преходи тип GOTO в или извън цикъла
  * Рекурентности (Recurrences)
  * Зависимости по данни между отделните итерации
  * Сложно организирани цикли (Complex coding, difficult loop analysis)

▶ Флагове за отключване на векторизиращите възможности на компилатора

| Компилатор | SSE Flag | AVX Flag | Reporting |
|---|---|---|---|
| GNU | -O3 | -mavx<br>-mavx2 (C/C++ only) | -fopt-info-all |
| Intel | -vec (по подразбиране) | -axAVX<br>-axCORE-AVX2 | -qopt_report |
| PGI | -Mvect=simd:128 | -Mvect=simd:256 | -Minfo=all |

## Проблеми на автовекторизацията: *пример*

```
1  void add_floats(float *a, float *b, float *c, float *d,
   float *e, int n)
2  {
3      int i;
4      for (i=0; i<n; i++)
5      {
6          a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
7      }
8  }
```

Цикъл, който не може да бъде автовекторизиран от компилатора. ***Причина***: Множеството указатели, предавани към функцията; Възможно е презастъпването им в паметта.

```
1  void add_floats(float *a, float *b, float *c, float *d,
   float *e, int n)
2  {
3      int i;
4  #pragma omp simd
5      for (i=0; i<n; i++)
6      {
7          a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
8      }
9  }
```

Ако като разработчици сме уверени, че няма опасност цикълът да бъде векторизиран, то с помощта на директива simd това може да бъде задействано – **направлявана автовекторизация**