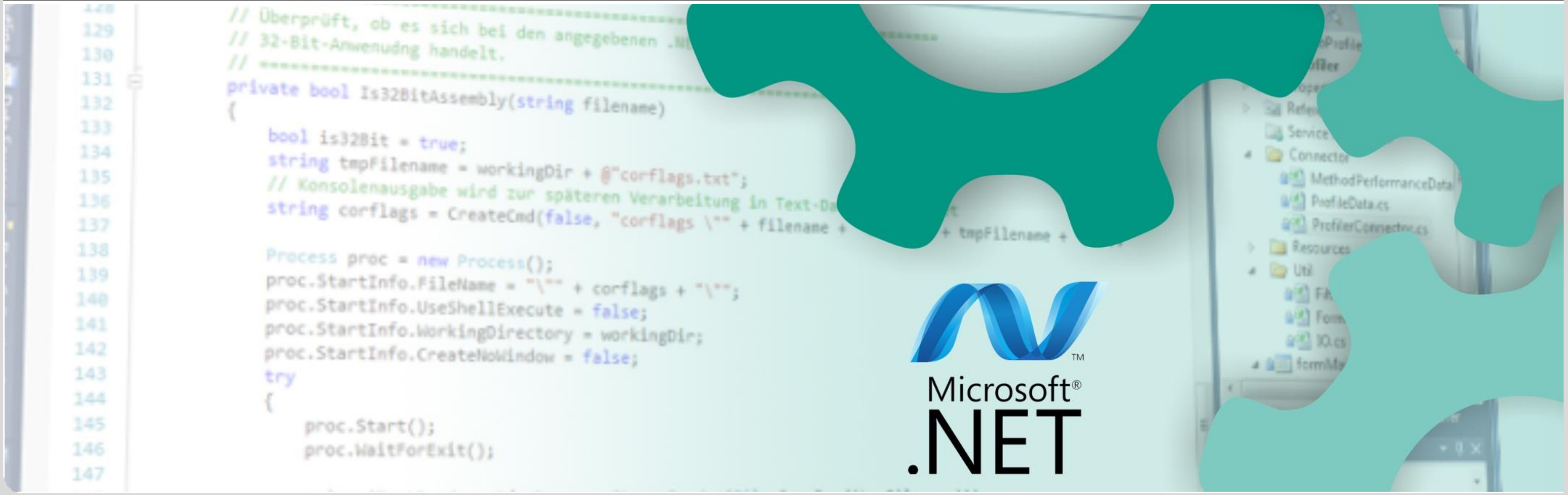
# The multicore era has started. Are 40 years of sequential source code obsolete?

**3rd FITTEST Industrial Day, 7th IEEE International Conference on Research Challenges in Information Science (RCIS 2013, Sorbonne, Paris)**

**Korbinian Molitorisz**

IPD Tichy – Programming systems chair



```
// Überprüft, ob es sich bei den angegebenen .N
// 32-Bit-Anwenudng handelt.
// ==========================================
private bool Is32BitAssembly(string filename)
{
    bool is32Bit = true;
    string tmpFilename = workingDir + @"corflags.txt";
    // Konsolenausgabe wird zur späteren Verarbeitung in Text-Da
    string corflags = CreateCmd(false, "corflags \"" + filename +        + tmpFilename +

    Process proc = new Process();
    proc.StartInfo.FileName = "\"" + corflags + "\"";
    proc.StartInfo.UseShellExecute = false;
    proc.StartInfo.WorkingDirectory = workingDir;
    proc.StartInfo.CreateNoWindow = false;
    try
    {
        proc.Start();
        proc.WaitForExit();
```

Corporate Technology

## Motivation

- Multicores are ubiquotious
- Parallel software skills are not
- Parallel software is neither
- The free lunch is over – clock frequency stagnation[1]

⇨ Do we all have to become parallel experts now?

- Refactoring support for existing software needed.[2]

⇨ Automation?

1) David Paterson, Herb Sutter (2006, 2009)

2) Hans Vandierendonck: *Averting the Next Software Crisis* (2011)

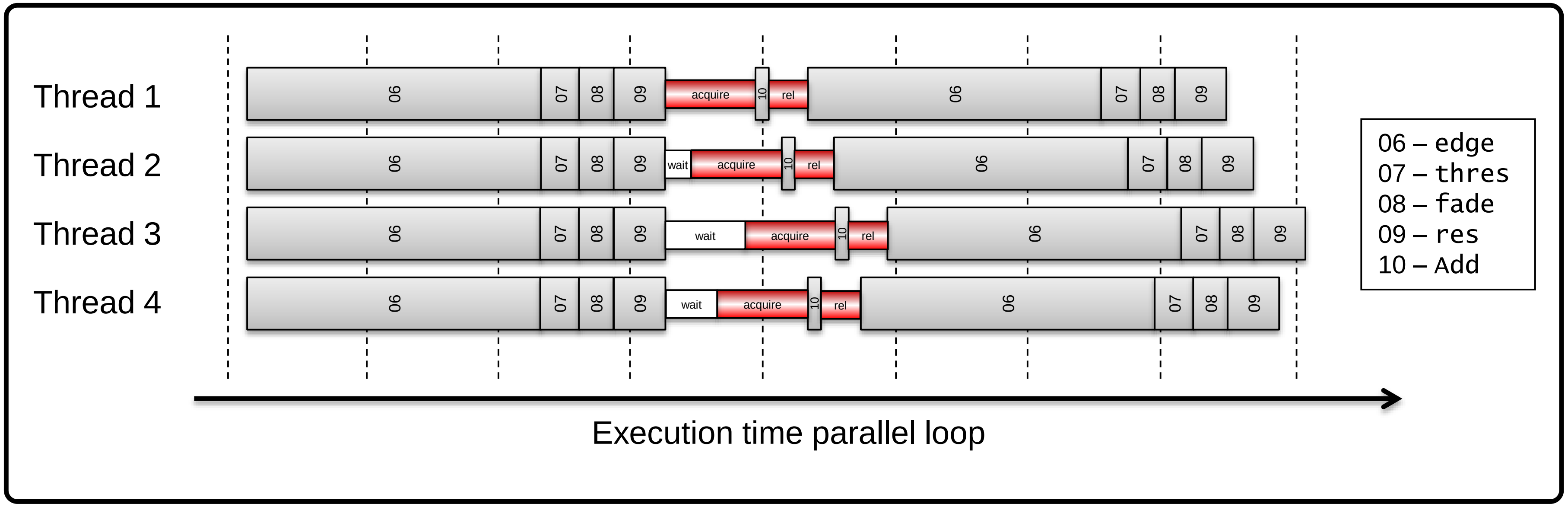
## Motivation

```
AviStream Process(AviStream aviIn)
{
 AviStream aviOut = new AviStream();
 foreach(Image i in aviIn.Images)                       1.440x
 {
  Image edge  = edgeFilter.Apply(i);                       68%
  Image thres = thresholdFilter.Apply(i);                   9%
  Image fade  = fadingFilter.Apply(i);                      8%
  Image res   = addFilter.Apply(edge, thres, fade);        12%
  aviOut.Images.Add(res);                                   3%
 }
 return aviOut;
}
```

➡ Gold standard: Parallel loop?

AForge.NET – *Library for Parallel Programming*, http://www.aforgenet.com, 2013

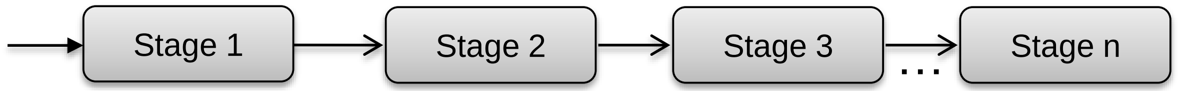
# Parallel loop



- Threads need to wait very long (aquisition/release of common used lock)
- Correct sequence not guaranteed without additional logic

➔ Can architecture patterns be used and derived automatically?

## Architecture pattern: Pipeline

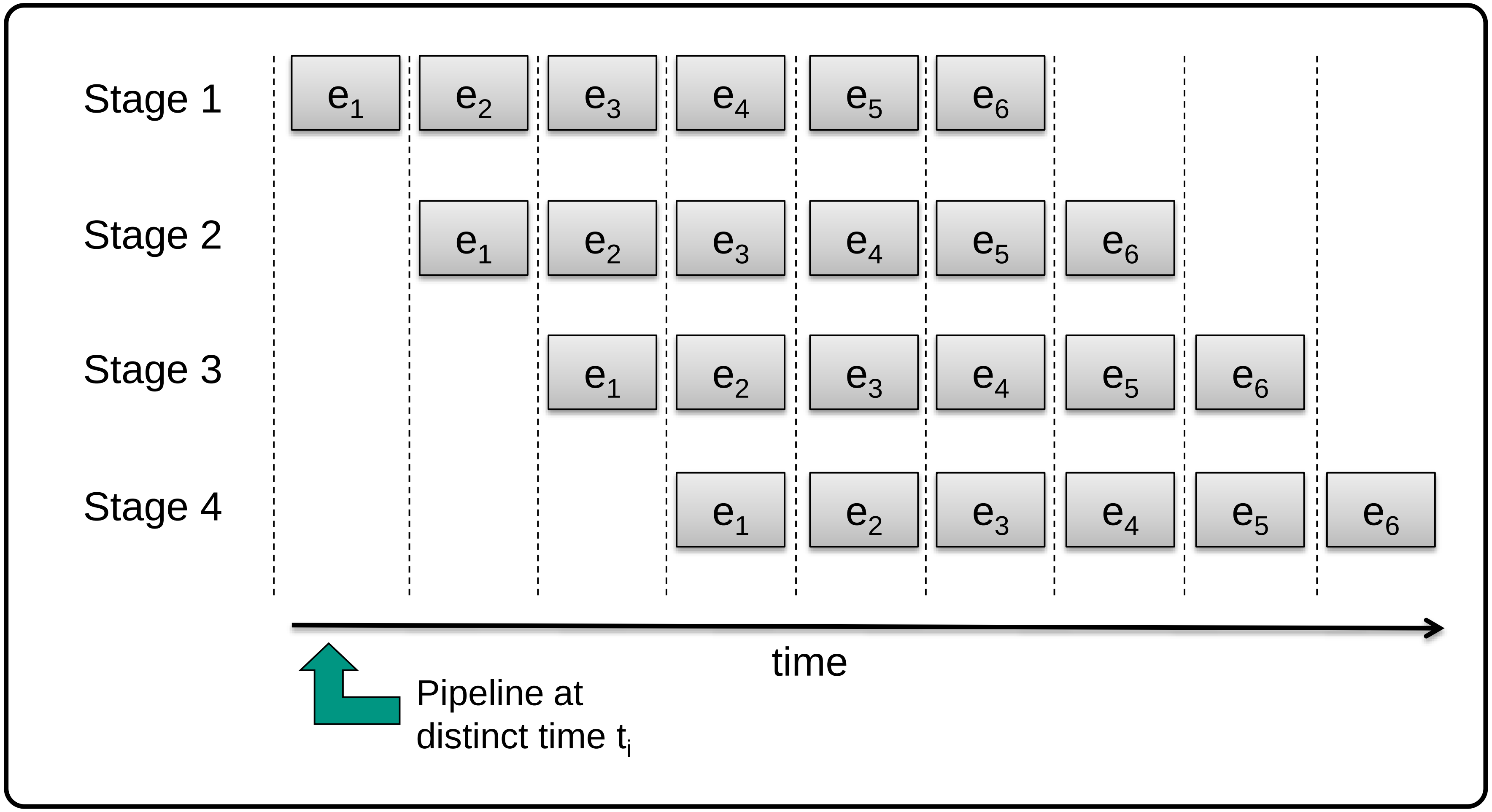


- Divide tasks into different stages that can be executed consecutively
- Dependencies between stages may exist (i.e. output of stage $s_i$ is the input for the following stage $s_{i+1}$)
  - Partly sequential execution, but dependencies within procedures and across iterations are preserved
- Data passes stages in a seqeuence
  - Might be cached between stages

Timothy G. Mattson, Beverly A. Sanders, Berna K. Massingill – *Patterns for Parallel Programming*, 2004

## Architecture pattern: Pipeline

- Illustration for data elements $e_1$ to $e_6$



Stage 1
Stage 2
Stage 3
Stage 4
$e_1$ $e_2$ $e_3$ $e_4$ $e_5$ $e_6$
$e_1$ $e_2$ $e_3$ $e_4$ $e_5$ $e_6$
$e_1$ $e_2$ $e_3$ $e_4$ $e_5$ $e_6$
$e_1$ $e_2$ $e_3$ $e_4$ $e_5$ $e_6$
time
Pipeline at distinct time $t_i$

## Observations

- Recurring patterns exist: Architecture / design / code access patterns…
  - ➡ Pattern-based approach

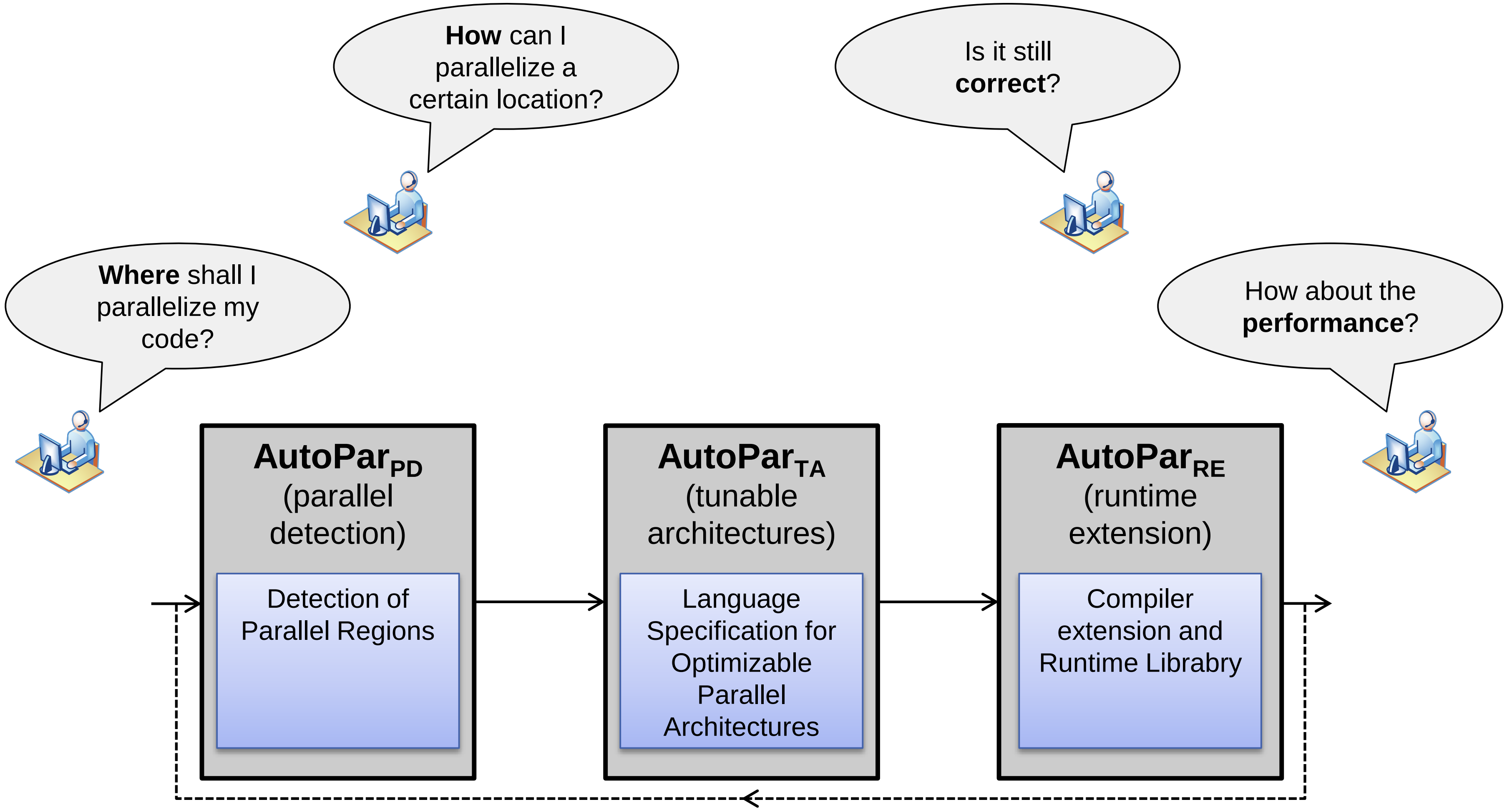
- Code areas consume different amounts of runtime
- Separation of concerns used in object-orientation
- Existing software builds on object-orientation
  - ➡ Split up control flow

- Modern object-oriented environments heavily use references
  - ➡ Combine static and dynamic analyses

- Parameters exist that have influence on the runtime behaviour
  - ➡ Derive tuning parameters from sequential runtime behaviour

- Race detectors exist but not handy for real-world applications
  - ➡ Unit tests as small fractions of a whole program

# Pattern-based refactoring concept: AutoPar



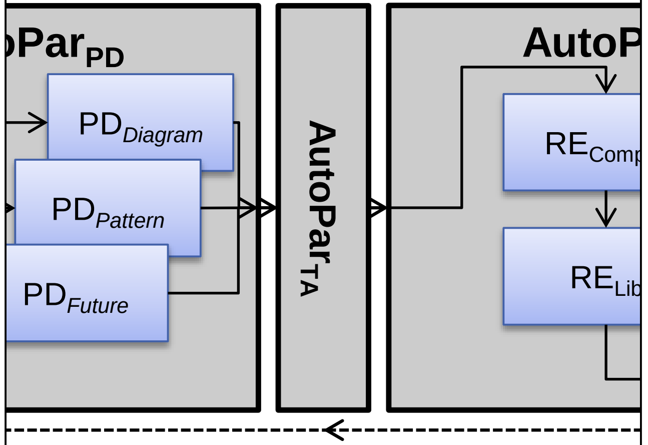
**Where** shall I parallelize my code?
**How** can I parallelize a certain location?
Is it still **correct**?
How about the **performance**?
AutoPar$_{PD}$ (parallel detection)
Detection of Parallel Regions
AutoPar$_{TA}$ (tunable architectures)
Language Specification for Optimizable Parallel Architectures
AutoPar$_{RE}$ (runtime extension)
Compiler extension and Runtime Library

Korbinian Molitorisz, Jochen Schimmel, Frank Otto – *Automatic Parallelization using AutoFutures*, MSEPT 2012
Jochen Schimmel, Korbinian Molitorisz, Ali Jannesari, Walter F. Tichy – *Automatic Generation of Parallel Unit Tests*, AST 2013

# Pattern-based refactoring concept: AutoPar

```
AviStream Process(AviStream aviIn)
{
 AviStream aviOut = new AviStream();

 foreach(Image i in aviIn.Images)    1.440x
 {
  Image e = edgeFilter.Apply(i);        68%
  Image t = thresholdFilter.Apply(i);    9%
  Image f = fadingFilter.Apply(i);       8%
  Image r = addFilter.Apply(e, t, f);   12%
  aviOut.Images.Add(r);                  3%
 }
return aviOut;
}
```



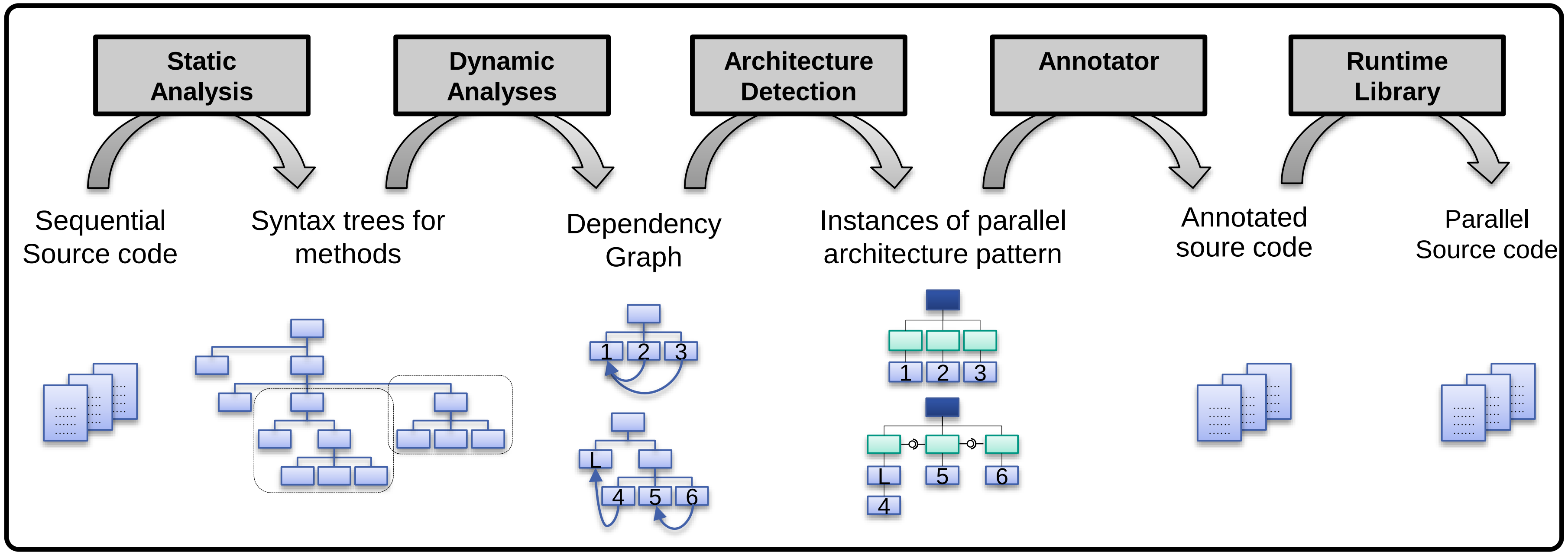
```
AviStream Process(AviStream aviIn)
{
 AviStream aviOut = new AviStream();
 #region TADL: (A+ || B || C) => D => E

   ForkJoin tasks = new ForkJoin();
   tasks.Add(edgeFilter, Tuning.Replicable);
   tasks.Add(thresholdFilter);
   tasks.Add(fadingFilter);

   Pipeline p = new Pipeline();
   p.Add(tasks);
   p.Add(addFilter);
   p.Add(stageE);
 #endregion
 return aviOut;
}
```

- Analysis pattern: *Single Static Multiple Dynamic*
- Detection modules operate on extended AST
- Explicit architecture language with tuning information
- Runtime library with stencils for patterns
- Automatic unit test generation
- Interface for auto tuners

## Pattern-based refactoring concept: AutoPar



- 5 separate steps to parallelize and test correctness in an automated process on the base of architecture patterns

# Pattern-based refactoring concept: AutoPar

- Code annotation
  - Architecture description language with definied operators (for architecture description) and operands (for the architecture compartments)
- Runtime library
  - Input: Architecture description and architecture compartments
  - Output: Instances of the runtime library, tuning file and unit tests

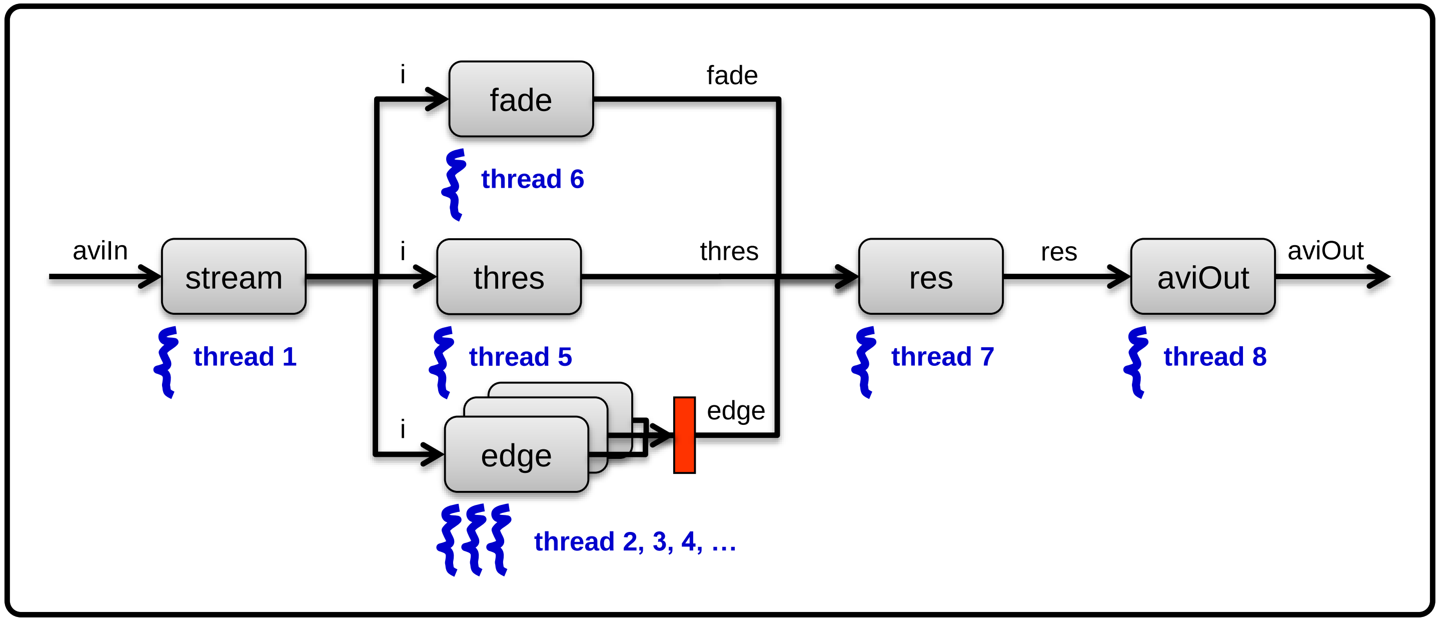
```
AviStream Process(AviStream aviIn)
{
 AviStream aviOut = new AviStream();
 #region TADL: (A+ || B || C) => D => E
 foreach(Image i in aviIn.Images)
 {
  #region A: Image e = edgeFilter.Apply(i);       #endregion
  #region B: Image t = thresholdFilter.Apply(i);  #endregion
  #region C: Image f = fadingFilter.Apply(i);     #endregion
  #region D: Image r = addFilter.Apply(e, t, f);  #endregion
  #region E: aviOut.Images.Add(r);                #endregion
 }
 #endregion
 return aviOut;
}
```

```
AviStream Process(AviStream aviIn)
{
 Item p1 = new Item (edgeFilter.Apply());
 Item p2 = new Item (thresholdFilter.Apply());
 Item p3 = new Item (fadingFilter.Apply());
 Item p4 = new Item (addFilter.Apply());
 Item p5 = new Item (aviOut.Images.Add());
 MasterWorker mw = new MasterWorker (p1, p2, p3);
 mw.Item(p1).replicable = true;
 Pipeline p = new Pipeline (mw, p4, p5);
 p.Input = aviIn.Images;
 p.Run();
 return p.Output;
}
```

## Back to the example…

- Results:
  - Speedup on an 8-core machine: 3,12
  - Manual implementation of this architecture pattern: 6,2

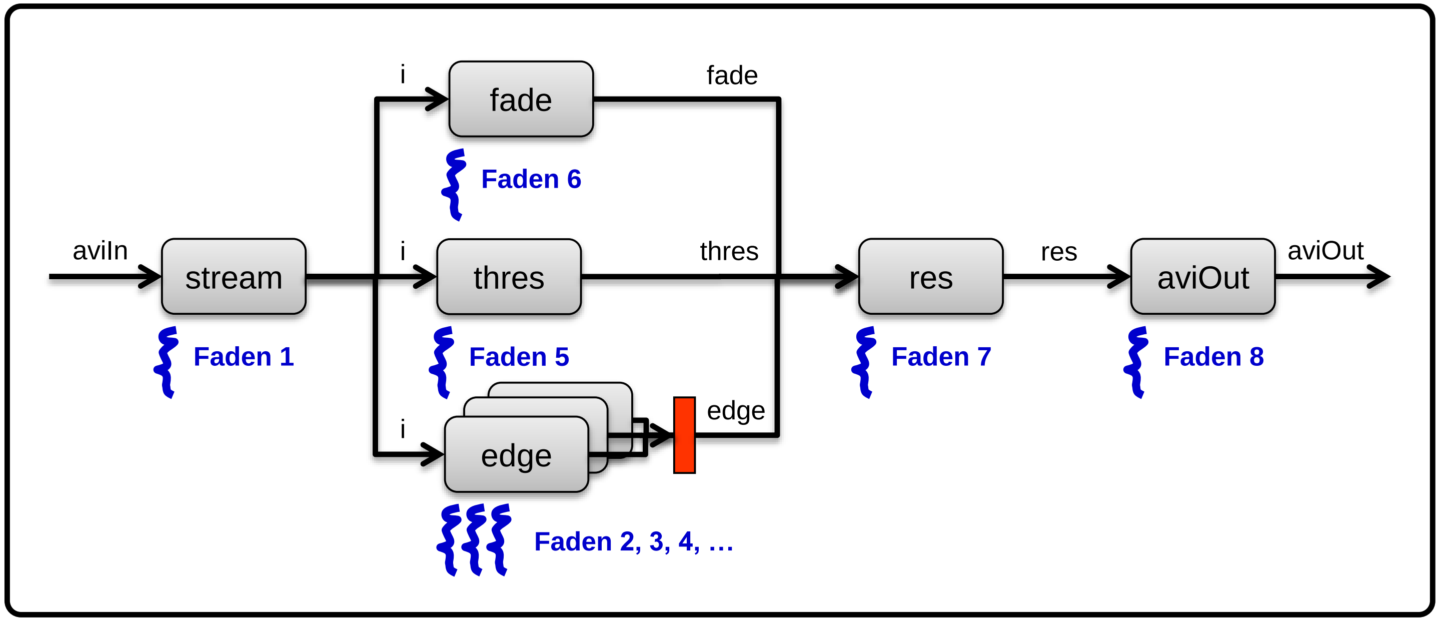
☹



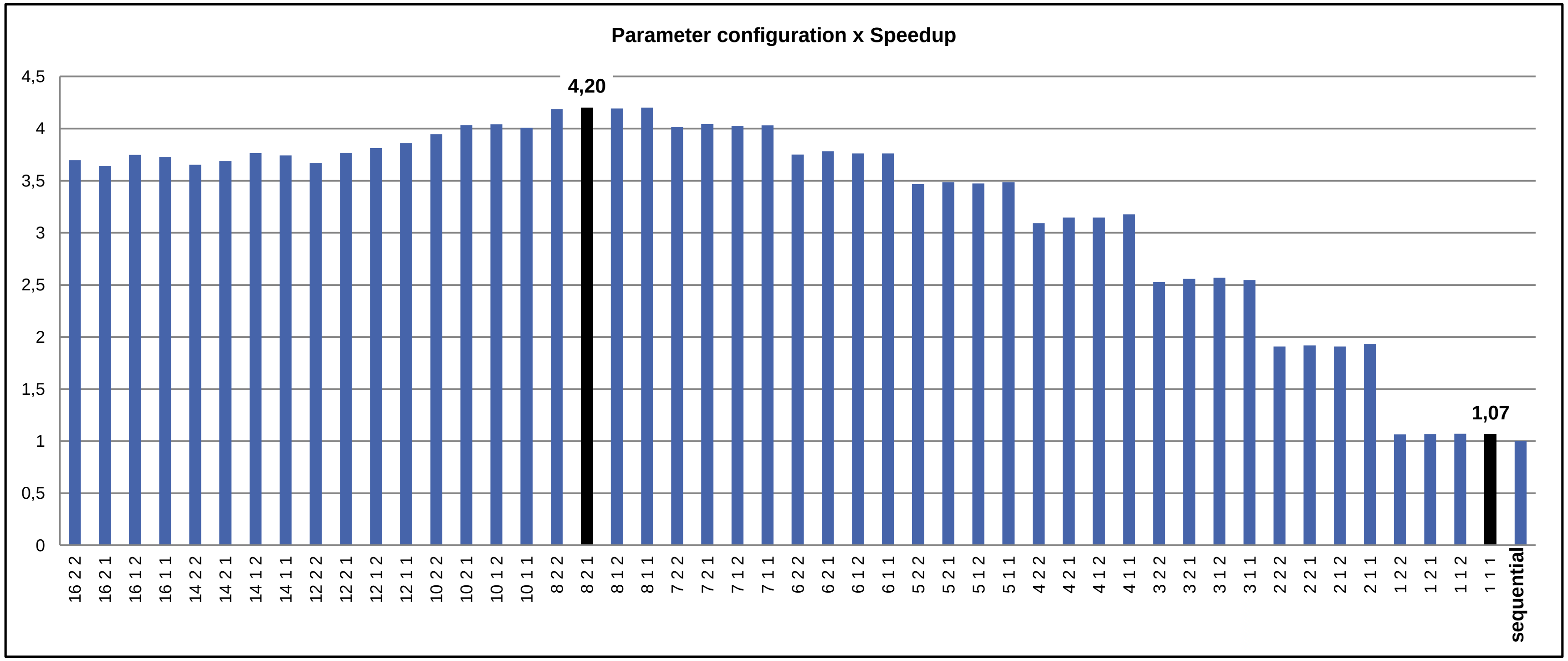
## Back to the example…

- Results:
  - Speedup on an 8-core machine: 5,3
  - Manual implementation of this architecture pattern: 6,2

☺



# Pattern-based refactoring concept: AutoPar



## Evaluation

- 6 real-world projects
- 27.000 LOC
- Average search space reduction: 95%
- Average precision: 66%

| Project | #HotSpots | #Architectures identified | Reduction | Correct & faster | Correct & ¬faster | ¬Correct | Precision |
|---|---|---|---|---|---|---|---|
| MergeSort | 19 | 1 | 95% | 1 | 0 | 0 | 100 % |
| RayTracer | 63 | 6 | 90% | 1 | 5 | 0 | 17 % |
| DesktopSearch | 47 | 2 | 96% | 1 | 1 | 0 | 50 % |
| CompGeo | 169 | 1 | 98% | 1 | 0 | 0 | 100 % |
| VideoProcessing | 16 | 1 | 94% | 1 | 0 | 0 | 100 % |
| PowerCollections | 3.641 | 15 | 97% | 4 | 9 | 2 | 27 % |
|  | ∑ = 3.955 | ∑ = 26 | Ø = 95% | ∑ = 9 | ∑ = 15 | ∑ = 2 | Ø = 66% |

Thank you for your attention.

Any questions?

molitorisz@kit.edu