Generative aspects in skeletal systems

M. Danelutto
Dept. Computer Science - Univ. of Pisa
&
CoreGRID Virtual Institute
on Programming Model

Summary

- Skeletons
- Classical skeleton implementation (template)
- Alternative implementation (mdf + rewrules)
- Layered implementation
- Conclusions

Summary

- Skeletons
- Classical skeleton implementation (template)
- Alternative implementation (mdf + rewrules)
- Layered implementation
- Conclusions

Skeletons

- Useful, parametric, efficient parallelism exploitation pattern
  - **useful** : for a large class of applications
  - **parametric** : in the seq code, parallelism degree, types of tasks and results
  - **efficient** : known efficient implementations on a range of architectures

Sample skel: farm

- **farm: ('a → 'b) → stream 'a → stream 'b**
  farm f <x1, ... , xn> = <f(x1), ... , f(xn)>
  f(xi) independent of f(xj) any i,j (parallel)
- **useful** most of currently large scale parallel application fit the schema
- **parametric** in code, data types and parallelism degree
- **efficient** master slave, SMP multithread, ...

Pros

- Separation of concerns
  - user: qualitative aspects
  - system: quantitative ones + mechanisms
- Portability (source code)
- Optimizations
  - exploit high level (meta) info
  - e.g. comm clustering, source rewriting, ...

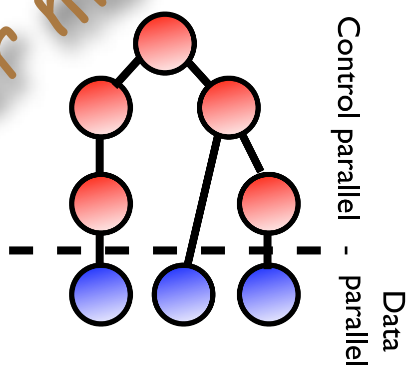
Cons

- fixed skeleton set
  - impossibility to exploit slightly different parallel patterns
  - impossibility to introduce new parallel patterns
- poor / no interoperability with other parallel frameworks

Typical skeletons

- Control parallel (aka stream parallel):
  - pipelines, task farms, loops, ...
  - parallelism : computation of different stream items
- Data parallel
  - map, reduce, prefix, ...
  - parallelism : computation of different (possibly overlapping) partitions of the same stream item

Nestable (two tier model)



Summary

- Skeletons
- Classical skeleton implementation (template)
- Alternative implementation (mdf + rewrules)
- Layered implementation
- Conclusions

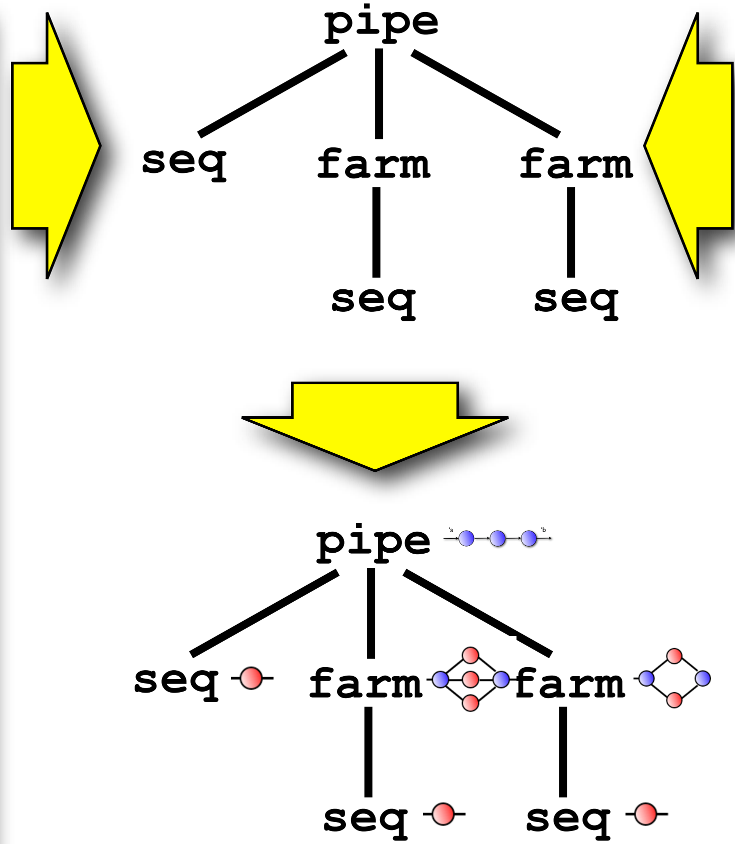
Generative aspects

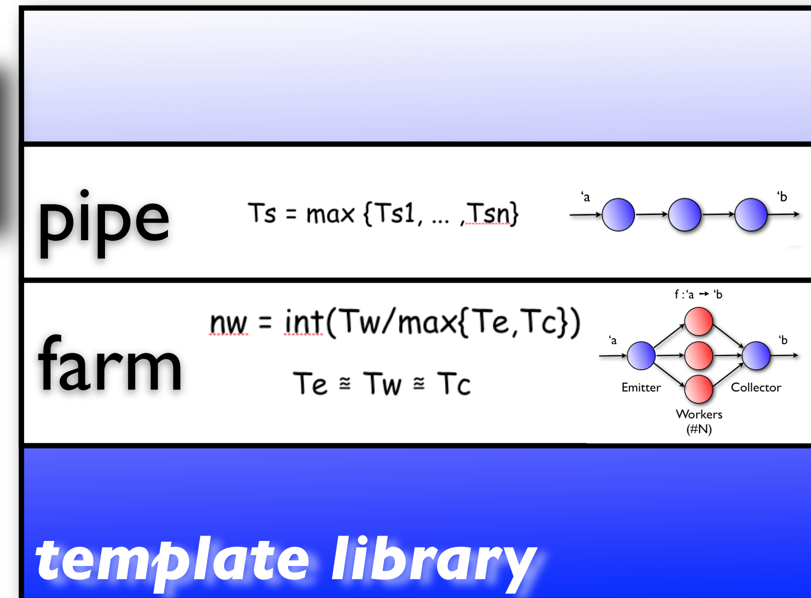
- Very diverse formalisms
  - skeletons : high level coordination patterns
  - object code : sequential language plus communication library calls
- Exploit existing knowledge and heuristics
  - to produce efficient, realistic obj code (template based implementations)

Template based impl

```
seq f in(...) out(...)
$c{ ... }c$

seq g in(...) out(...)
$c{ ... }c$

seq h in(...) out(...)
$c{ ... }c$

farm s2 in(...) out(...)
  g in(...) out(...)
end farm

farm s3 in(...) out(...)
  h in(...) out(...)
end farm

pipe main in(...) out(...)
  f in(...) out(...)
  s2 in(...) out(...)
  s3 in(...) out(...)
end pipe
```

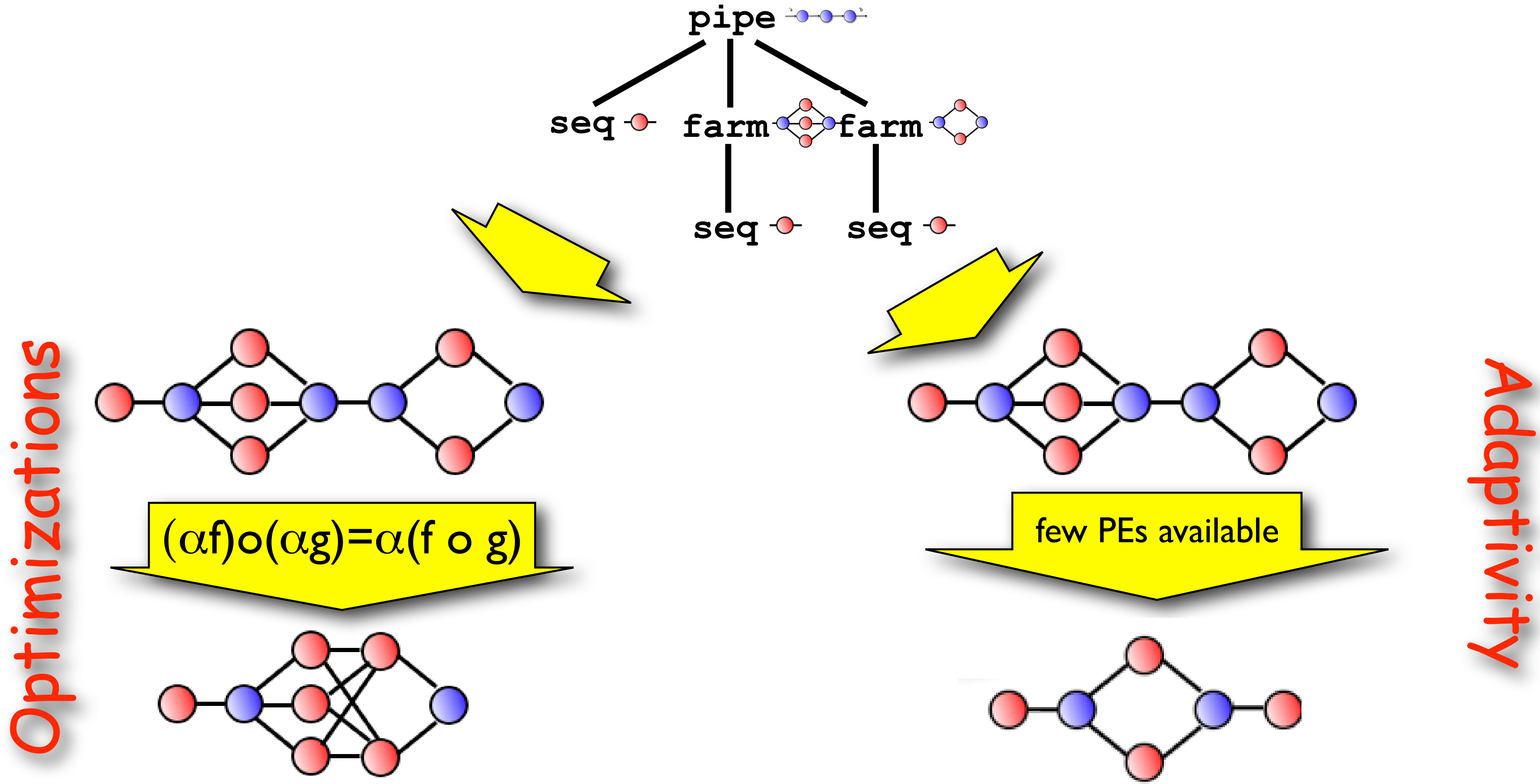
source code





this is P3L ('92)

Template based impl



Problems & issues

- proper mechanisms to implement templates
  - decided at compile time (target nodes?)
- performance models
  - need rough estimate of run/comm times (profiling? user supplied?)
- optimizations
  - need knowledge relative to target config

Template libs

- Template based, run time only implementations
- Library calls (user responsibility) to: define skel tree, invoke execution
- Library call implementation:
  - de facto instantiate templates
  - with additional parameters from users

this is Muesly ('02)

Summary

- Skeletons
- Classical skeleton implementation (template)
- Alternative implementation (mdf + rewrules)
- Layered implementation
- Conclusions

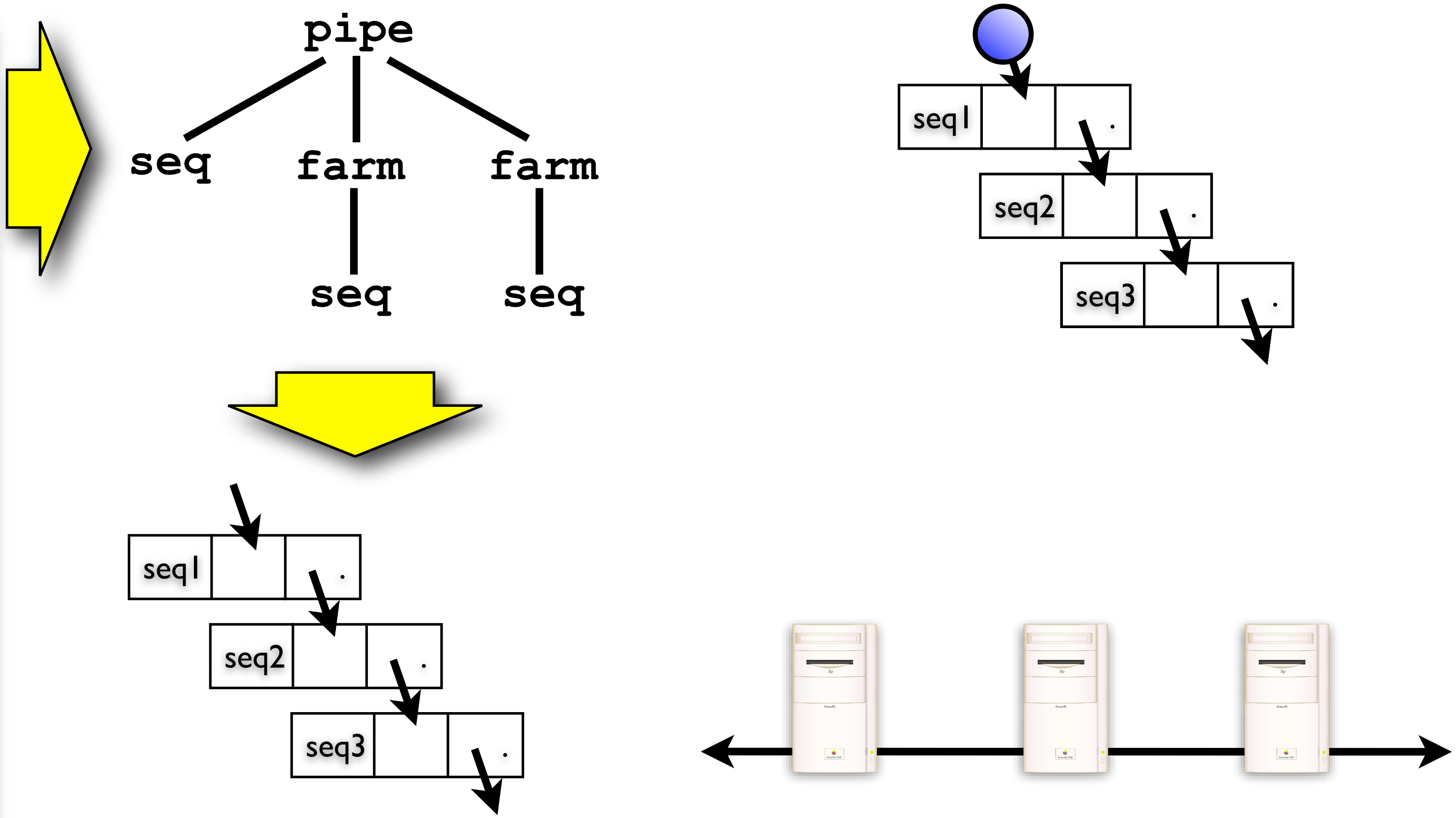
Generative aspects

- First skel generation (P3L, Muesli, eSkel)
  - code generated is virtually exposed to programmers (they do know its structure)
- Then: source2source transforms, optims, non template based implement. techniques
  - programmers unaware of what's going on
  - skeletons → meta info to be exploited

Macro data flow impl

```
import muskel.*;
public class Main {
public static void
   main(String[] args) {

Compute seq1 = new Seq1();
Compute seq2 = new Seq2();
Compute seq3 = new Seq3();
Farm s2 = new Farm(seq2);
Farm s3 = new Farm(seq3);
Pipeline farms =
 new Pipeline(s2,s3);
Pipeline main =
 new Pipeline(seq1,farms);
Manager mng =
 new Manager(main);
m.inputStream(“in.dat”);
m.outputStream(“o.dat”);
m.compute();
return;
}
```

source code

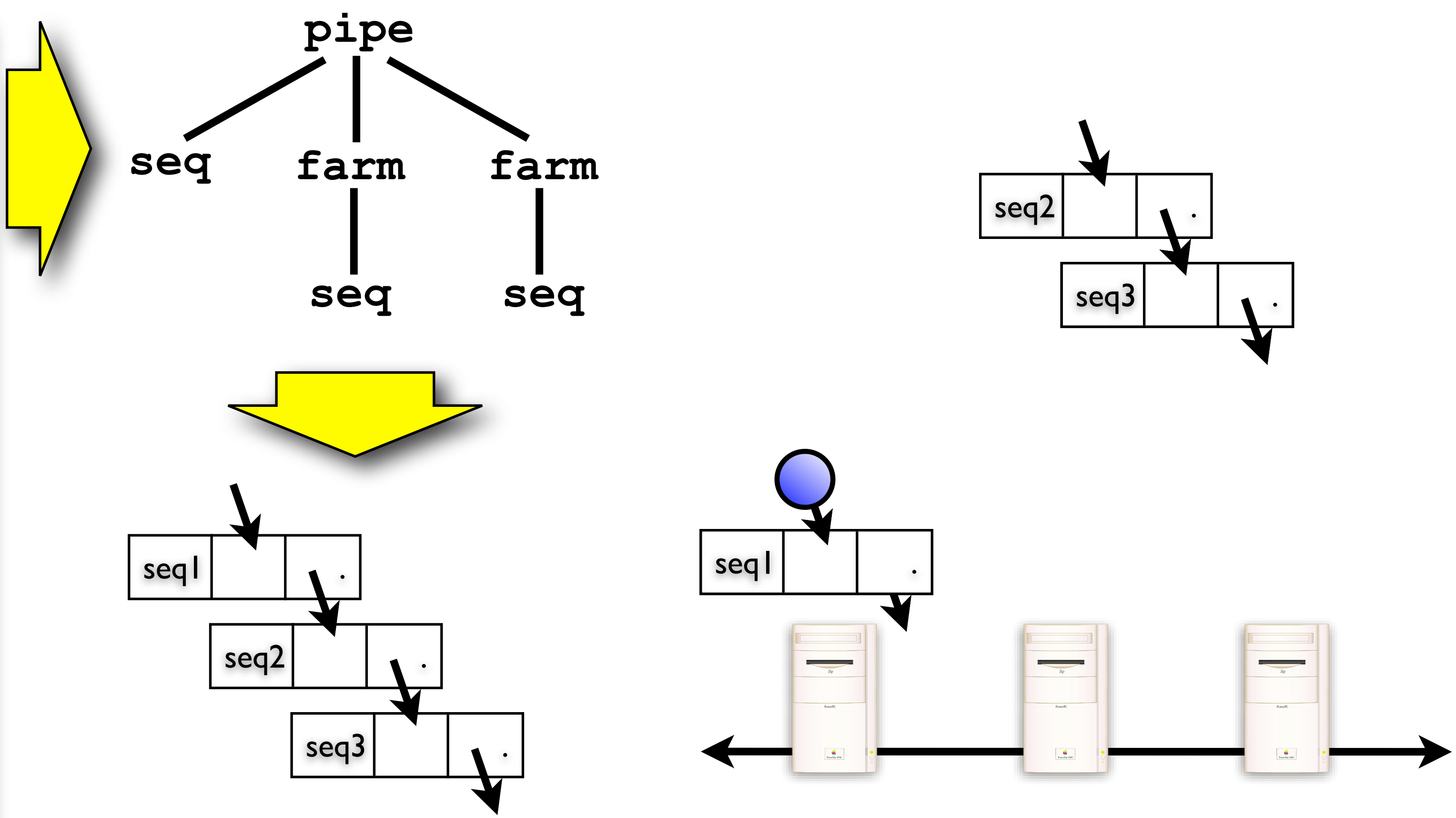


this is Lithium/muskel ('00)

Macro data flow impl

```
import muskel.*;
public class Main {
public static void
   main(String[] args) {

Compute seq1 = new Seq1();
Compute seq2 = new Seq2();
Compute seq3 = new Seq3();
Farm s2 = new Farm(seq2);
Farm s3 = new Farm(seq3);
Pipeline farms =
 new Pipeline(s2,s3);
Pipeline main =
 new Pipeline(seq1,farms);
Manager mng =
 new Manager(main);
m.inputStream("in.dat");
m.outputStream("o.dat");
m.compute();
return;
}
```

source code

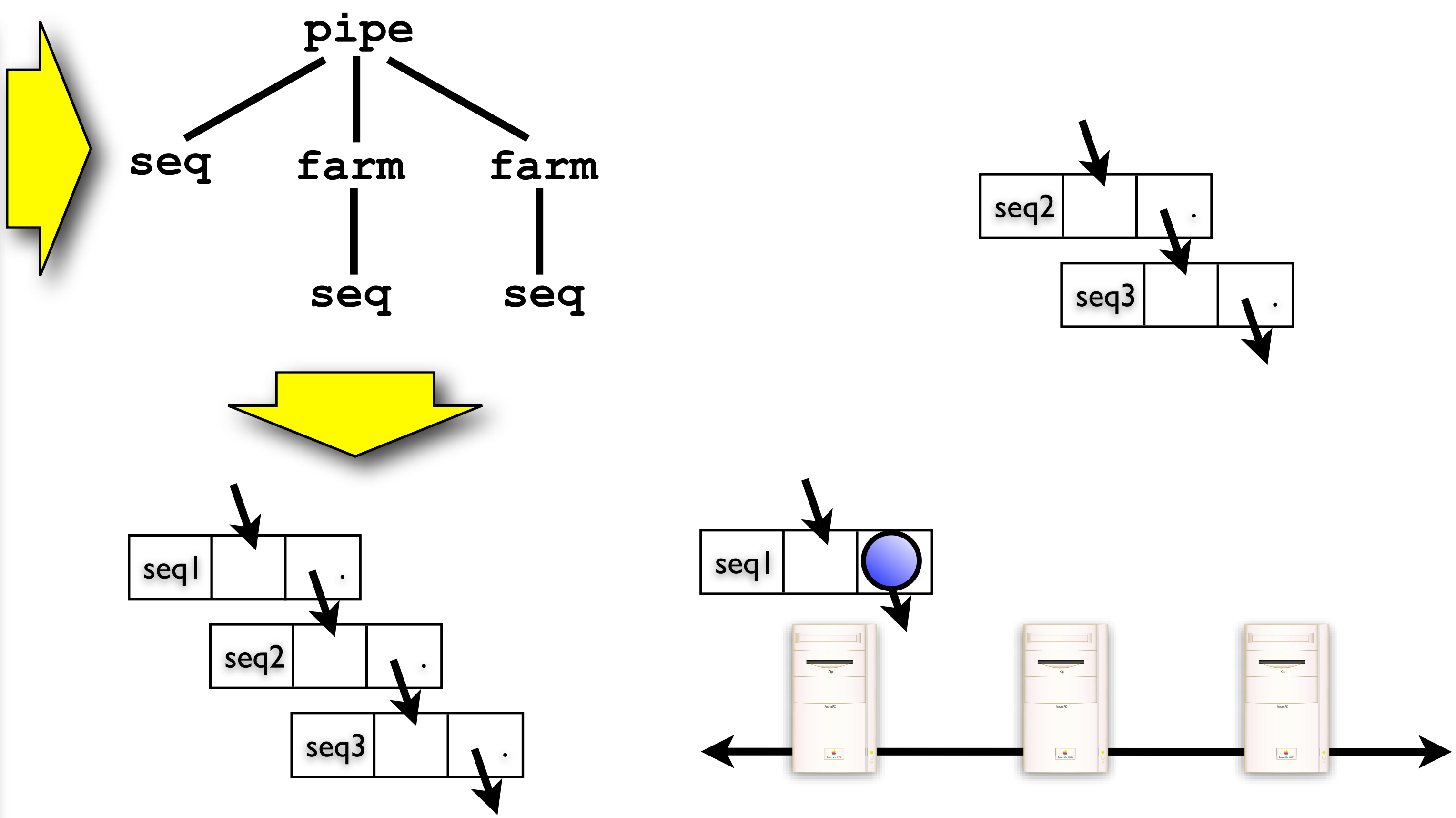


this is Lithium/muskel ('00)

Macro data flow impl

```
import muskel.*;
public class Main {
public static void
   main(String[] args) {

Compute seq1 = new Seq1();
Compute seq2 = new Seq2();
Compute seq3 = new Seq3();
Farm s2 = new Farm(seq2);
Farm s3 = new Farm(seq3);
Pipeline farms =
 new Pipeline(s2,s3);
Pipeline main =
 new Pipeline(seq1,farms);
Manager mng =
 new Manager(main);
m.inputStream("in.dat");
m.outputStream("o.dat");
m.compute();
return;
}
```

source code

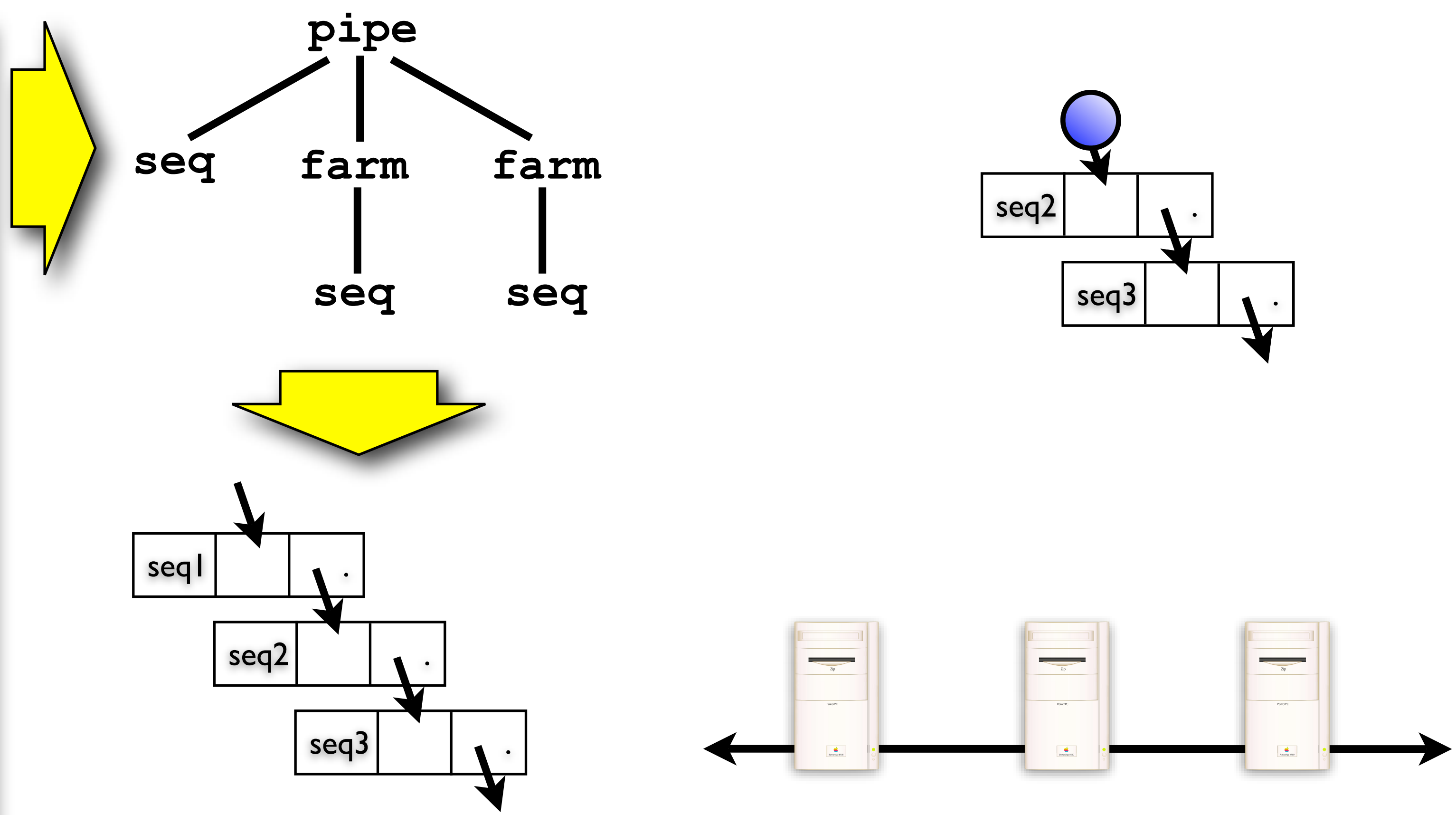


this is Lithium/muskel ('00)

Macro data flow impl

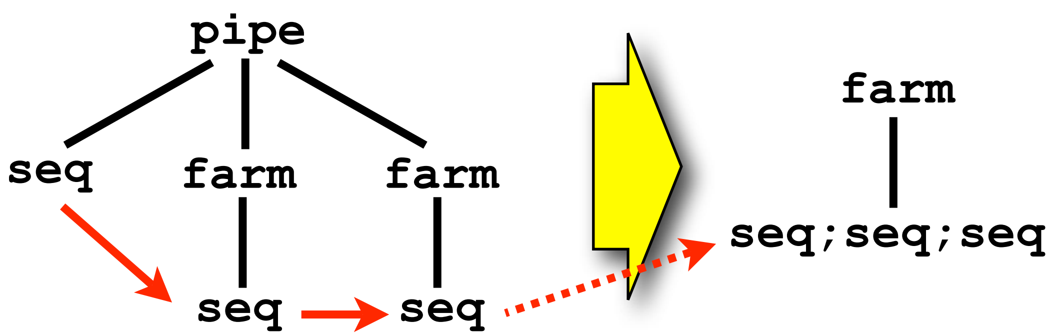
```
import muskel.*;
public class Main {
public static void
   main(String[] args) {

Compute seq1 = new Seq1();
Compute seq2 = new Seq2();
Compute seq3 = new Seq3();
Farm s2 = new Farm(seq2);
Farm s3 = new Farm(seq3);
Pipeline farms =
 new Pipeline(s2,s3);
Pipeline main =
 new Pipeline(seq1,farms);
Manager mng =
 new Manager(main);
m.inputStream("in.dat");
m.outputStream("o.dat");
m.compute();
return;
}
```

source code



this is Lithium/muskel ('00)

Normal form ('99)



Skel = Seq | farm(Skel) | pipe(Skel,Skel)

- NF(Seq) = Seq
- NF(farm(Skel)) = NF(Skel)
- NF(pipe(Skel',Skel") = NF(Skel');NF(Skel")
- normalForm(Skel) = farm(NF(Skel));

+ increases computational grain
+ decreases comm overhead
+ improves service time
+ uses leq amount of PEs

MDF + NF

- Locality handled by support rather than programmer
  - affinity scheduling of MDF instructions
- Grain handled by the support
  - communication clustering / multithreaded remote interpreters
- Better skeleton sets to be investigated (meta annotations vs. skeletons)

Summary

- Skeletons
- Classical skeleton implementation (template)
- Alternative implementation (mdf + rewrules)
- Layered implementation
- Conclusions

Generative aspects

- Layered implementation of structured parallel programming environments
- Each layer with precise responsibilities
  - local optimizations
- Strictly hierarchical layer layout
  - avoid local optimization interference

Layers

Applications
Qualititative parallelism exploitation, user knowledge

Compiler tools
Static code generation + static transforms/optims, system designer knowledge

Loader / Deployment tools
Target machine dependent code choice, system designer + machine expert knowledge

Run time system
Adaptivity, fault tolerance, QoS, ... , machine expert knowledge

Layered systems

```
generic main()
{
 stream T1 s1;
 stream T2 s2;

 stage1(output_stream s1);
 stage2(input_stream s1,
   output_stream s2);
 stage3(input_stream s2);
}

proc s1(output_stream T1
s) $c{ /* C code */ }c$

stage1(output_stream T1 s)
{
  s1(output_stream s)
}

parmod stage2(input_stream
T1 sin, output_stream T2
sout) {
```

source code

astCC

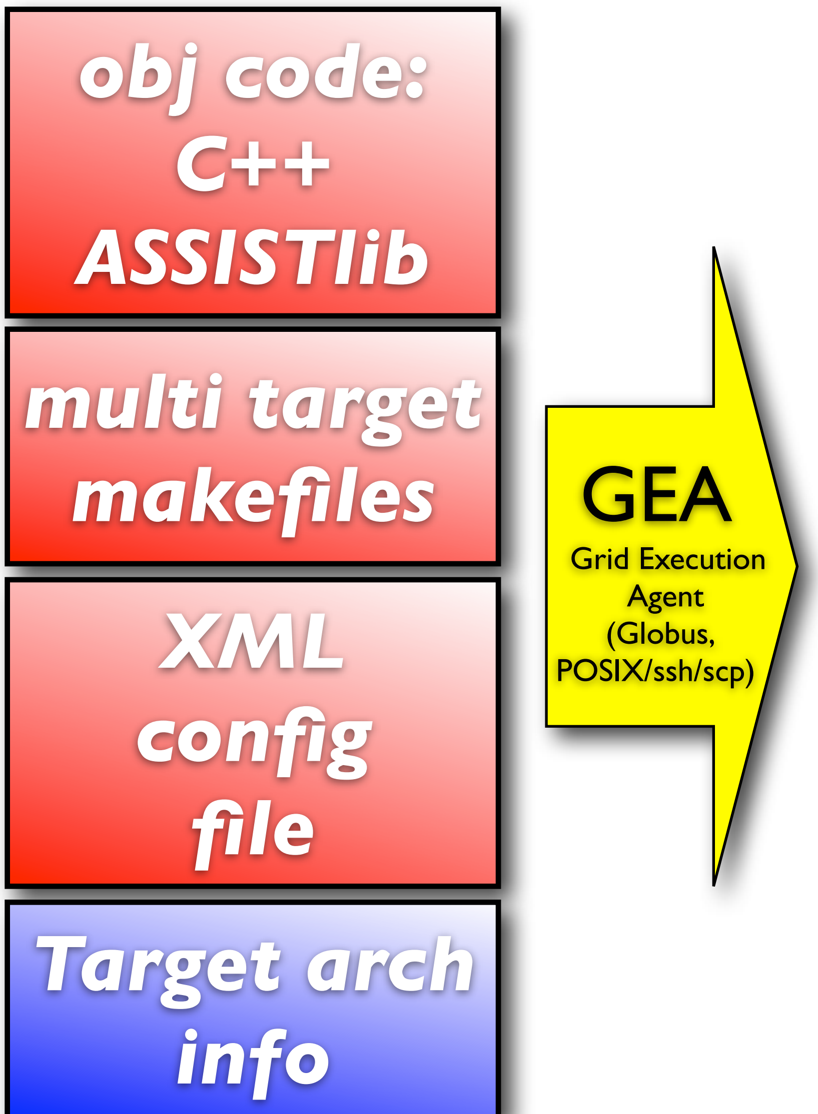
obj code: C++ ASSISTlib — ready to compile code

multi target makefiles — relative to diff target archs

XML config file — sw config (libs, run time) + proc network config

this is ASSIST ('00)

Layered systems

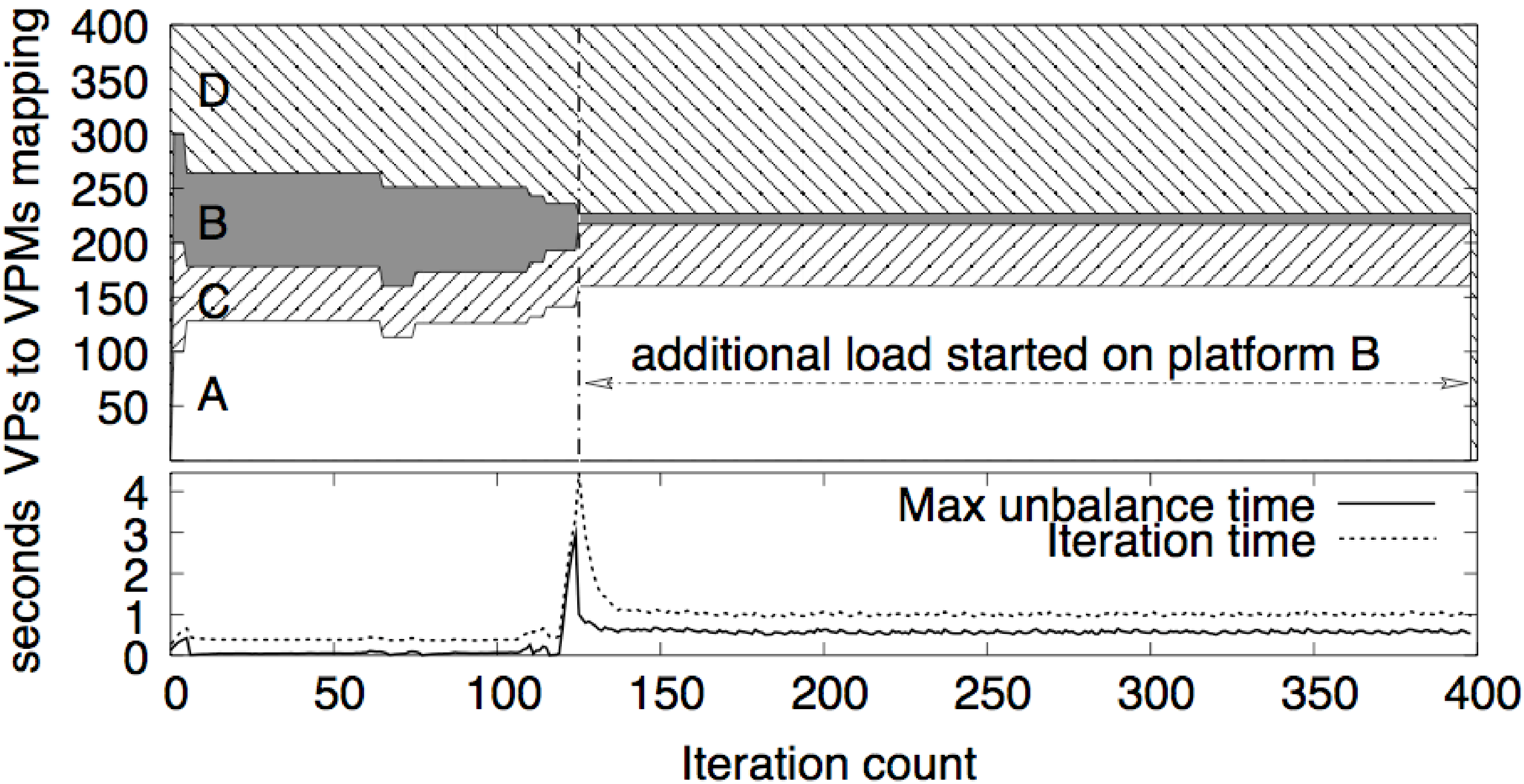


+ figures out target machine features
+ compiles proper code subsets using proper makefiles
+ deploys/stages user and system code
+ starts up run time system processes, user (derived) processes, management processes
+ stages input data
+ starts and monitors program execution
+ stages back output data

Plus ...

- interoperability
  - compiler wraps ASSIST code to WS/CCM
- fault tolerance
  - comp. generated control code works at run time
- adaptivity
  - compiler generates manager code
  - manager adapts prog behavior at run time

ASSIST adaptivity



Summary

- Skeletons
- Classical skeleton implementation (template)
- Alternative implementation (mdf + rewrules)
- Layered implementation
- Conclusions

Conclusions

- Skeletons provide very high level information, exploitable to generate good code
- Separation of concerns is fundamental to provide efficiency
- Proper choice of compile/load/run time timing is fundamental as well

- any questions ?

(marcod@di.unipi.it)

References

- P3L: Susanna Pelagatti "Structured Development of Parallel Programs" Taylor&Francis '98
- muskel: www.di.unipi.it/~marcod
- ASSIST: www.di.unipi.it/Assist.html

- marcod@di.unipi.it