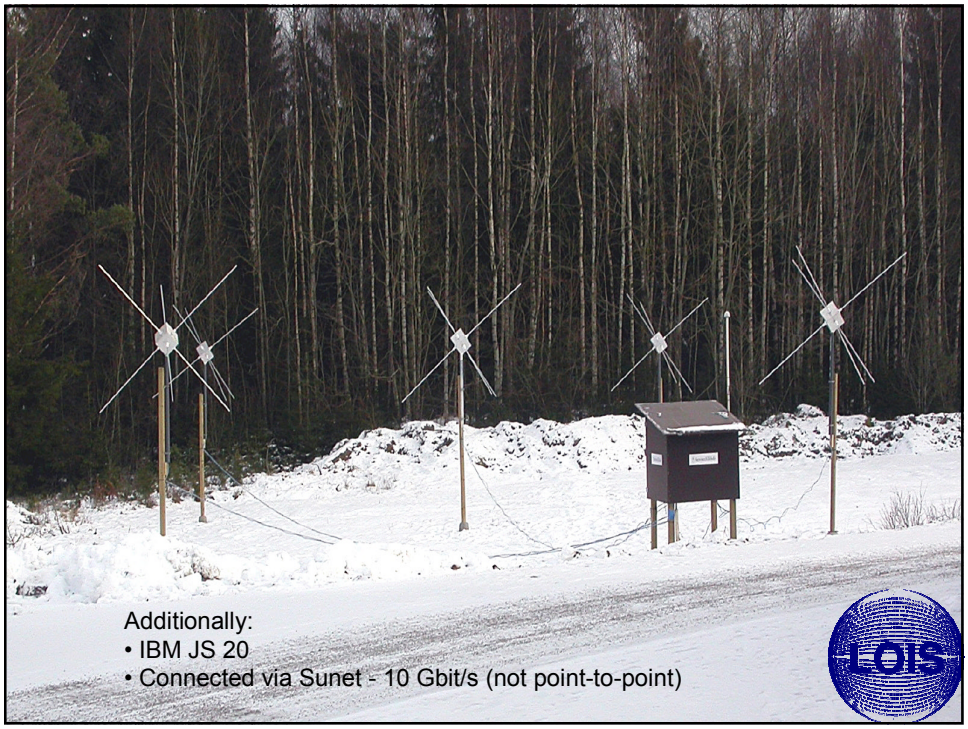
A Grid-like Infrastructure for Sensor-Networks

J. Andersson
M. Ericsson
M. Karlsson
Welf Löwe

Växjö University

www.LOIS-Space.net Project

- Multi-purpose radio research facility,
- Primarily intended for radio signal based astrophysics
  - to study transient phenomena Earth's space environment
  - to discover and monitor long-term trends
- 13,500 digital radio receiver units, together working like a huge parabolic antenna
  - each producing data at a rate of 2 Gbits/s
- Receivers are equipped with custom processors together working like a supercomputer
  - ~40 Tflops computational power distributed across the stations
- Additionally: network of workstations, supercomputers
- Needs high-performance computing



Change Happens

- Hardware configuration:
  - Receivers are added/removed/changed
  - New computers are added/removed/changed
  - Network changes
- Deployed software:
  - Applications can be added / removed
  - QoS of these applications may change: precision, data rate, response time
- Needs flexibility

Users

- Users / programmers are scientists
- No trained software engineers
- Think in terms of their problem solution, in numerical algorithms (in the best case)
- Needs right level of abstraction for good programmability

This is a Problem

- High performance
- High flexibility
- Good programmability

- Cannot expect the optimum of all qualities

The Sensor GRID

- A *sensor-network* (hardware)
  - Set of sensors with synchronization hardware generating a stream of input values (three 16bit complex numbers)
  - Set of (heterogeneous) computational nodes for processing sensor-stream applications
  - Interconnection network
- An *stream application* (software)
  - Filter: a set of stream filters with data-dependencies
  - Service: a stream sinks, data-services
  - Components thereof
  - QoS parameters
- A *stream processing infrastructure* (runtime environment)
  - Communication
  - Optimization, load balancing
  - (un-)deployment

Outline

- How to get performance?
- How to add flexibility?
- How to add programmability?
- Open issues

Outline

- How to get performance?
- How to add flexibility?
- How to add programmability?
- Open issues

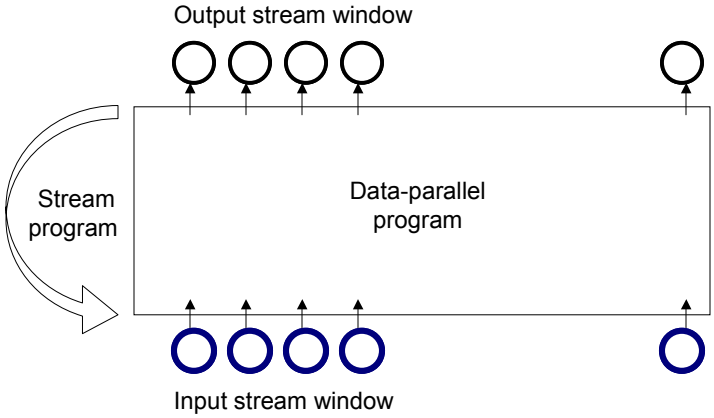
Static View

- One non-changing high-performance application
- Antennas send streams of time-stamped values
  - Beam forming: integration of values with same time stamp to a stream of signals
  - Buffering: collecting subsequent signals to window (or split)
  - Processing streams values
- Then applications consist of
  - Stream filters
    - Pipe filter architecture with input stream data as source
    - Push driven communication between filters
    - Optimization goal data rate (sometimes completion time)
  - Stream services
    - Sinks of the filter architecture
    - Pull driven communication between services
    - Requires that the streams have integrated sufficiently - not time critical

Stream filters

- Data-parallel program
  - Input and output: stream window with time stamp
  - Notion of synchronism since signals come with time stamps
  - Virtual shared memory:
    - 2D array *Sensor* x *Time*
    - Contain sensor values over time
  - Fixed input size
    - Fixed number of sensors
    - Fixed size of windows splitting values over time
- Stream program:
  - Iteration over data-parallel programs
  - Possibly different input and output data rates

Stream filter (schema)



Optimization

- Series of Transformations
  - Data-parallel stream filters
  - Task graph for the data-parallel program
    - Asynchronous
    - Distributed memory
  - Cyclic schedule for the task graph
    - Objective function: usually data rate
    - Heterogeneous computational nodes in the sensor network
- Deployment to the sensor network

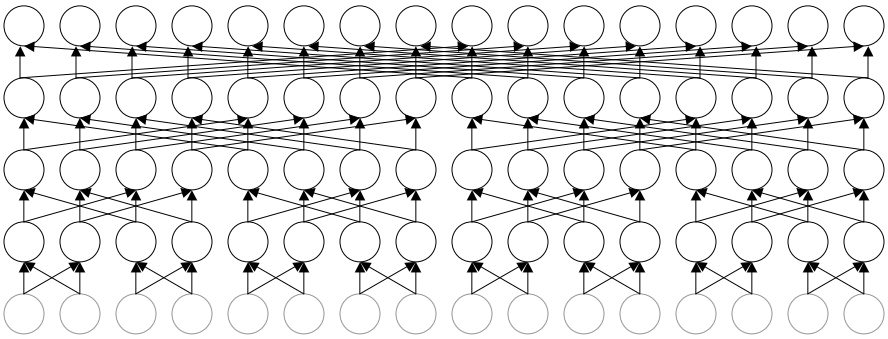
Example: FFT

fun fft(v:stream array[n] of complex):
stream array[n] of complex

```
//r(i) denotes the value of the reversed bit representation of i.
for i=0…n-1  do in parallel
    v[i] :=v[r(i)];
end;
for j=0…log n-1  do
    for k:=0…n/2^(j+1)-1; i:=0…2^j-1  do in parallel
        v[k*2^(j+1)+i] := v[k*2^(j+1)+i] + ω^(i*n/2^(j+1))* v[(2k + 1)*2^j+i];
        v[(2k + 1)*2^j+i] := v[k*2^(j+1)+i] - ω^(i*n/2^(j+1))* v[(2k + 1)*2^j+i];
    end;
od;
```

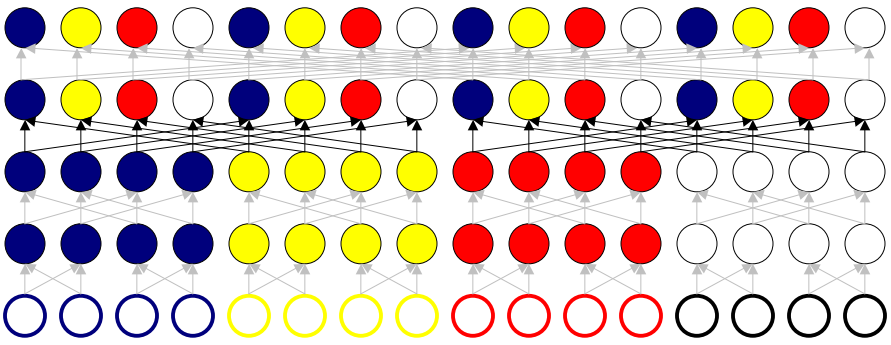
14

Task Graph: FFT



Input size 16

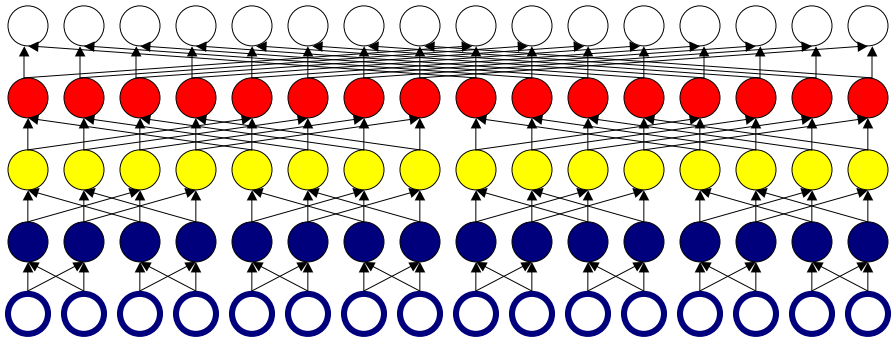
Schedule: FFT



Completion time: 16 x comp + 3 x send(1) + 3 x recv(1) + Latency

Data rate: 1 / (16 x comp + 3 x send(1) + 3 x recv(1) + Latency)

Schedule: FFT (alternative)



Completion time: 3 x (16 x comp + send(32) + recv(32) + Latency)

Data rate: 1 / (16 x comp + send(32) + recv(32))

Dynamic View

- User triggered, GRID portal
  - Adding or removing components,
  - Changing QoS requirements
- Application triggered
  - E.g. solar eruption
  - A probe component recognizes a pattern in the input stream requiring reconfiguration
  - Changing QoS enabling a new component
- System triggered
  - Imbalanced load,
  - Failure of hardware,
  - Added / changed hardware

Outline

- How to get performance?
- How to add flexibility?
- How to add programmability?
- Open issues

User Triggered Changes

- Sensor-GRID portal: different user groups submit their applications each aiming at the highest possible data rate
- Several applications reuse primitive, close to sensor components
- Each application can be optimized separately but what's the global optimum?
  - Build a global application and optimize it. Some user application might have satisfied its a minimum data rate
  - Campaigns: one application at a time. Some user groups get a time frame without interesting events.
- Approach: Market based optimization

Market based optimization

- Instantiation of a general optimization framework
- The market represents the global interests
  - Sets an initial price per processor and adjusts it
  - Computes a global by merging local schedules from user groups
  - Selects an optimum global schedules satisfying a subset of user groups
- Agents a_i represent user groups/applications i
  - a_i compute a set of schedules $s_{i,p=1..P}$ with data rate $f(s_{i,p})$
  - a_i has a utility function mapping data rates to a value $\$(f(s_{i,p}))$
  - a_i submits a pair <$\$, $s_{i,p}$> to the market
  - a_i has a strategy on how to react if a schedule $s_{i,p}$ cannot be satisfied (change $\$ and/or p)
- Then iteration over bidding and market decision until fixpoint found

Fix point iteration

- The following strategy terminates in a fix point provided there hold some simple condition on
  - utility functions and bidding strategy of agents
  - pricing strategies of market

1. Set an initial price per processor
2. Repeat
3. All a_i submit bids, i.e. pairs <$\$, $s_{i,p}$>
4. The market select the global schedule and processor price optimizing the value of bids satisfied (market makes surplus per value of bids satisfied)
5. Set new price and communicate global schedule to a_i
6. Until no more surplus increment can be found

Application Triggered Changes

- Changes triggered by application should be effective immediately (short reaction time)
- To achieve final performance, we need to apply static scheduling algorithms online
  - Performance of the scheduling determines the reaction time
  - Long reaction time since complex scheduling
- Idea: all application triggered events are known in advance
- Approach: look-ahead scheduling, i.e. prepare for possible changes before they are triggered

Look-Ahead Scheduling

- Distinguish between a set of conceptual application models, A, and their physical implementations, I,
- Scheduling, M, maps from $a \in A$ to $i \in I$, $i = M(a)$.
- An application triggered change event e causes a transition from a to $a' = t(e, a)$,
- Given a set of such events, E, it is possible to determine all possible changes to the baseline architecture, and their implementations
  - $A'(E, a) = \{ a' \mid a' = t(e, a), e \in E \}$
  - $I'(E, a) = \{ i' \mid i' = M(a'), a' \in A' \}$
- This constitutes the Look-Ahead(1) schedule.
  - Changes triggered by application are effective immediately
  - Time for static scheduling not on the critical pathe
  - Only determinates the rate of which events can be accepted.

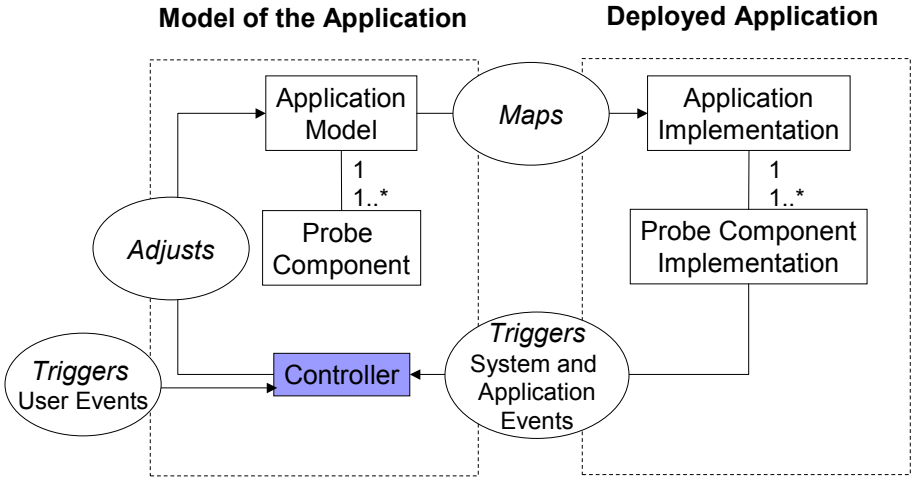
Changes revisited

- User triggered
  - Market based scheduling, then
  - Deploying the whole new application (easy since stateless applications),
  - Eventually more efficient: deploying the changed parts
  - Changes the application specification in the first place (model of the application)
- Application Triggered
  - Look-ahead scheduling
  - Changes the deployed application
- System triggered
  - Load balancing
  - Changes the deployed application

Consistency Problem

- Changes the application specification (model of the application) trigger changes the deployed application
- Changes the deployed application need to be propagated back to the application specification (model of the application)
  - Otherwise, user triggered changes update an outdated model
  - System and application triggered changes in between two user triggered changes disappear
- Standard problem to get dynamism in a static architecture
- Architectural pattern “dynamic architecture”

Dynamic Architecture



Outline

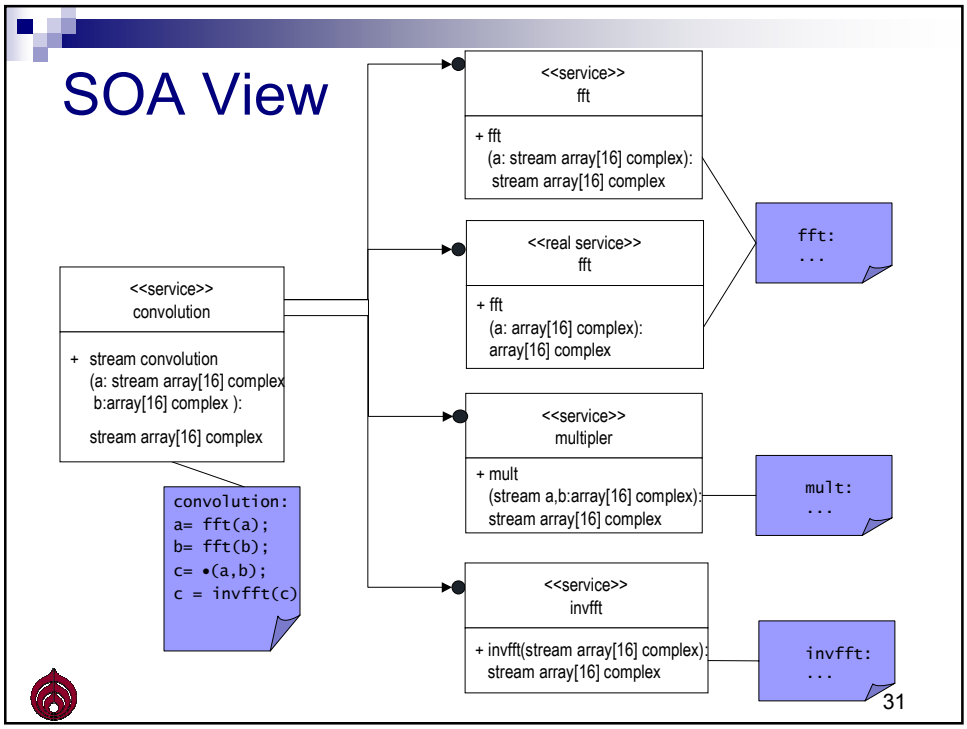
- How to get performance?
- How to add flexibility?
- How to add programmability?
- Open issues

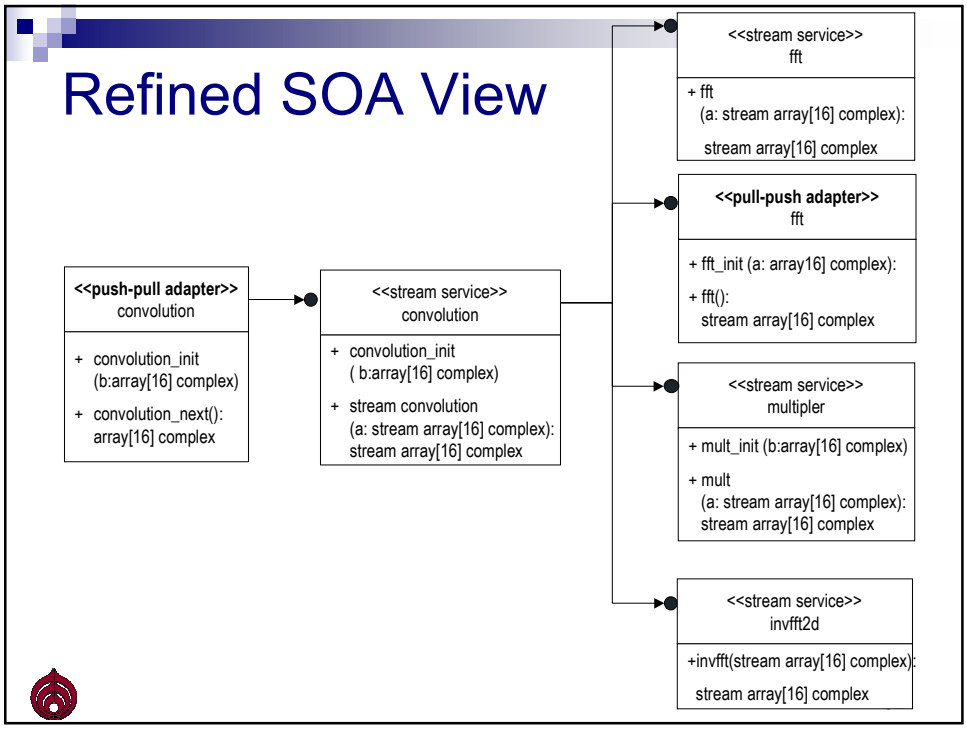
Service Oriented View

- Common SOA description layer on top of stream filters and stream services
  - Components and whole application provide services
  - Basic services provide input data
  - Request / Respond communication
- No clash of architecture when mixing the architectural styles visible to the programmer
- Easy composition
- Easy to integrate external services, e.g. data bases accesses
- Meet view of end user expecting a data service rather than a data stream

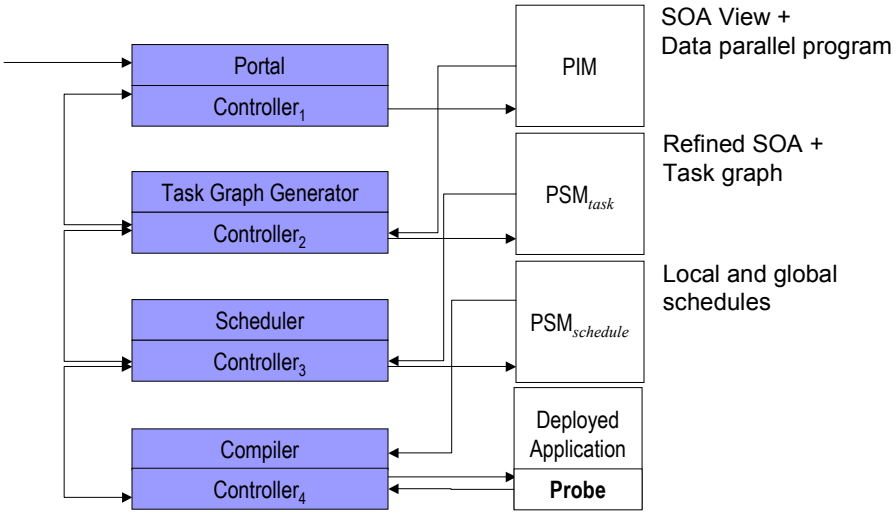
Model Driven Architecture – MDA

- Service view: UML description where different kinds of services modeled with new UML stereotypes:
  - Real service (imported or exported services “real” service)
  - Source services (encapsulates sensors)
  - Services (others)
- Transformed to refined UML descriptions
  - Service vs. stream implementation of components
    - Real services remain services (stream sinks)
    - Source services become data pumps (stream sources)
    - Other services can be implemented as either or
  - Buffers to adapt architectural mismatches, integrators/filters to adjust different data rates
- Transformation to data-parallel stream applications and further down


SOA View
<<service>>
fft
+ fft
(a: stream array[16] complex):
stream array[16] complex
<<real service>>
fft
+ fft
(a: array[16] complex):
array[16] complex
fft:
...
<<service>>
convolution
+ stream convolution
(a: stream array[16] complex
b:array[16] complex):
stream array[16] complex
convolution:
a= fft(a);
b= fft(b);
c= •(a,b);
c = invfft(c)
<<service>>
multipler
+ mult
(stream a,b:array[16] complex):
stream array[16] complex
mult:
...
<<service>>
invfft
+ invfft(stream array[16] complex):
stream array[16] complex
invfft:
...
31

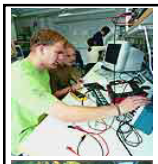

Refined SOA View
<<stream service>>
fft
+ fft
(a: stream array[16] complex):
stream array[16] complex
<<pull-push adapter>>
fft
+ fft_init (a: array16] complex):
+ fft():
stream array[16] complex
<<push-pull adapter>>
convolution
+ convolution_init
(b:array[16] complex)
+ convolution_next():
array[16] complex
<<stream service>>
convolution
+ convolution_init
(b:array[16] complex)
+ stream convolution
(a: stream array[16] complex):
stream array[16] complex
<<stream service>>
multipler
+ mult_init (b:array[16] complex)
+ mult
(a: stream array[16] complex):
stream array[16] complex
<<stream service>>
invfft2d
+invfft(stream array[16] complex):
stream array[16] complex

Dynamic Architecture + MDA



Open issues

- Program models above task graphs
  - Filters to adjust different data rates
  - Validate programmability
- System triggered change events
  - Integrate load balancing and
  - Map back load balancing decisions to the models
- Scheduling for multiple QoS requirements
  - Adequate machine models / scheduling techniques for heterogeneous sensor networks
  - Performance of scheduling algorithm
- Putting the loose strings together
- Implementation





Växjö University

Software Tech Group

- Software Analyses & Visualization
- Software Architecture & Composition
- 10 PhD students
- Young group, more than half of them joined last two years