Skeleton programming

Programming model Institute - CoreGRID
Dept. Computer Science - Univ. Pisa - Italy

Marco Danelutto

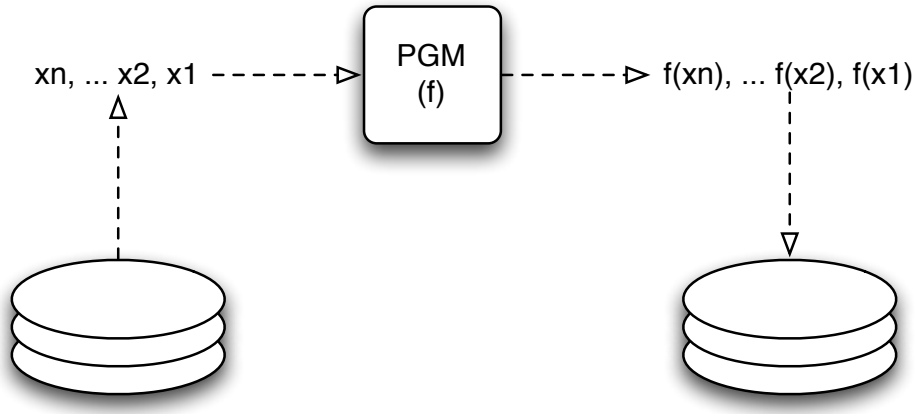
marcod@di.unipi.it

Contents

- State of the art grid programming
- Structured programming
- Skeletons
- Skeleton implementation (on grids)
- Behavioral skeletons (autonomic management)
- Conclusions

Sample application (UC1)

- **Use Case 1: *embarrassingly parallel application***
  - file with set of input data sets (tasks)
  - to be computed by means of program pgm (f)
  - results stored back to a file
  - variants:
    - tasks from the same data set
    - tasks from comm channels



UC1: qualifying features (grid!)

- input data available at one site
  - experiment location, satellite receiver, DBMS, ...
- results needed at one side
- coarse grain *f*
  - time spent computing >> time spent communicating
- lots of computing resources "*f enabled*" available

UC1 the “grid aware” way

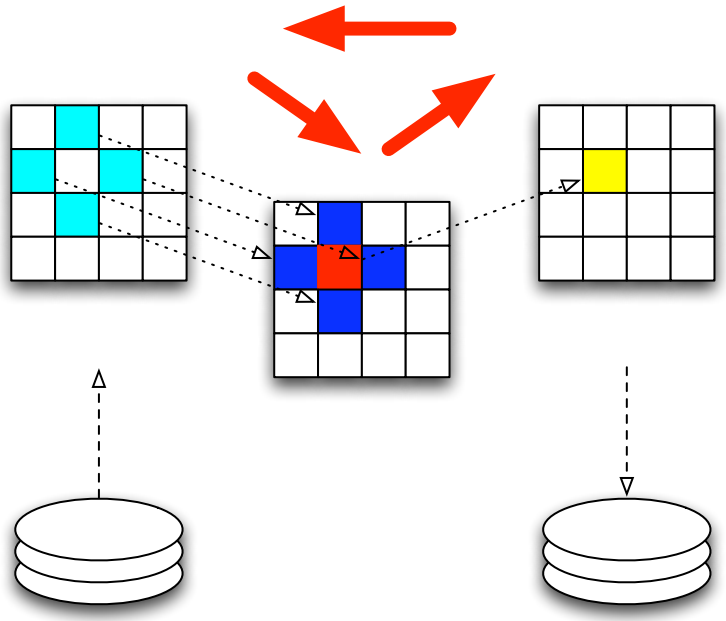
- functional concern
  - code for *f*, code retrieving input tasks and storing final results
- non functional concerns
  - resource recruitment
  - code staging and scheduling
  - data staging (to and from remote PEs)
  - explicit communication/synchronization handling

UC1 the “grid aware” way (2)

- non functional concerns (2)
  - security
    - data and code staging, code execution
  - load balancing
    - static and dynamic concern (PE power and PE load)
  - fault tolerance
    - model, implementation
  - efficiency

Sample case (UC2)

- **Use Case 2: *stencil data parallel application***
  - input data set: regular or irregular *collection*
  - iterative processing:
    - new item(i) = f(item(i), neighborhood(i))
    - check convergence (local + global)
  - store result: regular or irregular new collection

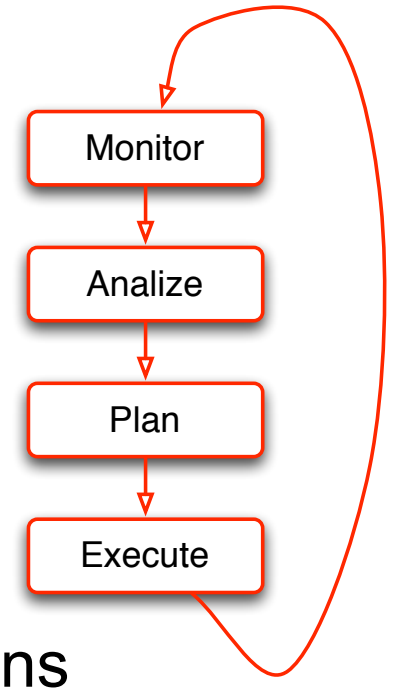


UC2 the “grid aware” way

- additional non functional concerns
  - co-{mapping, scheduling}
  - at each loop iteration
    - synchronization
    - data exchange
  - tradeoff
    - grain partitioning

UC3: autonomic adaptation to irregularity

- Irregularity sources
  - irregular applications, non dedicated target hw, faults, ...
- Properly handling:
  - recognize misbehaviors
  - adopt corrective policies
  - plan policy implementation
  - eventually implement planned actions



Alternative approach

- **Structured parallel programming**
  - provide programmers with pre-defined, customizable, optimized, possibly composable *parallelism exploitation patterns*
  - programmers: build applications using the pattern building blocks & provide seq hooks and code parameters to the patterns
  - system (compiler + run time): take completely care of all non functional aspects

Key idea (1)

- Separation of concerns (functional-non functional)
  - at the very beginning (late ‘80)
    - qualitative parallelism expressed by programmers strictly exploited “as is”
  - later one (early ‘00)
    - programmer qualitative parallelism partially drives/ suggests the actual implementation
    - tool optimizations can radically change the parallelism exploitation pattern used

Key idea (2)

- *Restricted* parallelism exploitation pattern set
  - restrict building blocks to *effective* ones
    - in insulation
    - when composed
  - *effective*
    - efficient implementation known
    - efficient composition known
    - efficient optimization/adaptation policies known

Key idea (3)

- skeleton language
  - or
- skeleton library

- exploit state of the art skeleton/pattern implementation

Typical skeletons/parallel patterns

- Stream/control parallel
  - pipeline, farm, loops
- Data parallel
  - map, reduce, prefix, stencil, ...
- Other
  - divide&conquer, parmod

Typical Grid applications

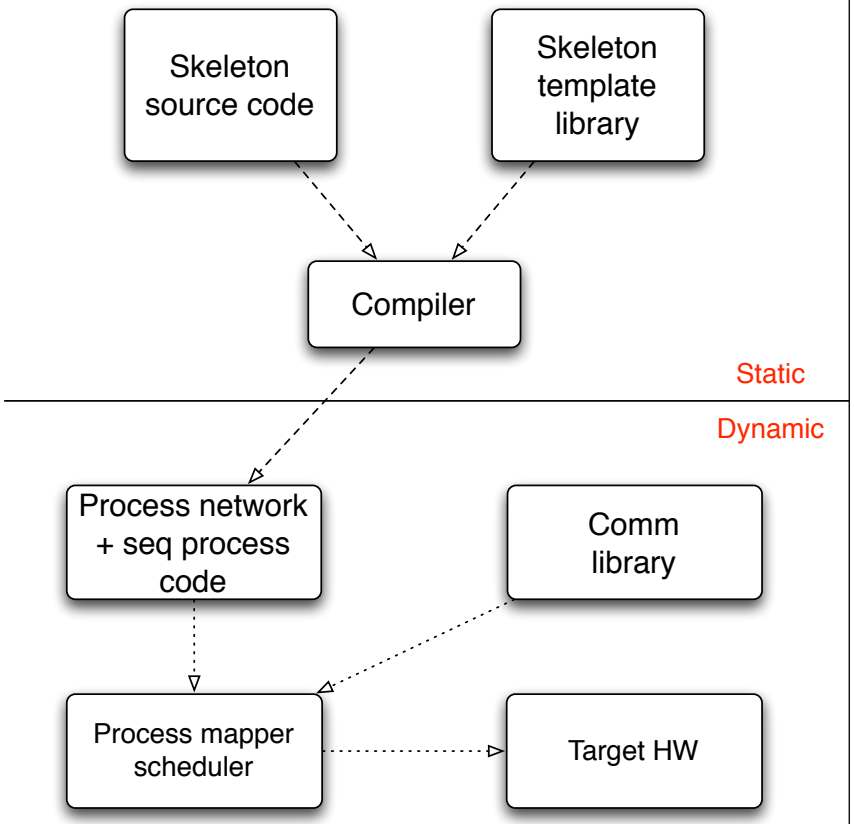
- UC1: embarrassingly parallel
  - pipe(taskGenerator,
    farm(f),
    resultSink)
- UC2: data parallel (stencil)
  - loop(
    pipe(map(gather),
    map(f),
    map(reduce)))

Tradeoffs

- large vs. small skeleton set
- general purpose vs. domain specific skeletons
- single vs. multiple host language supported
- library vs. language
- skeletons vs. patterns
- fixed set vs. expandable (user defined skeletons)
- ...

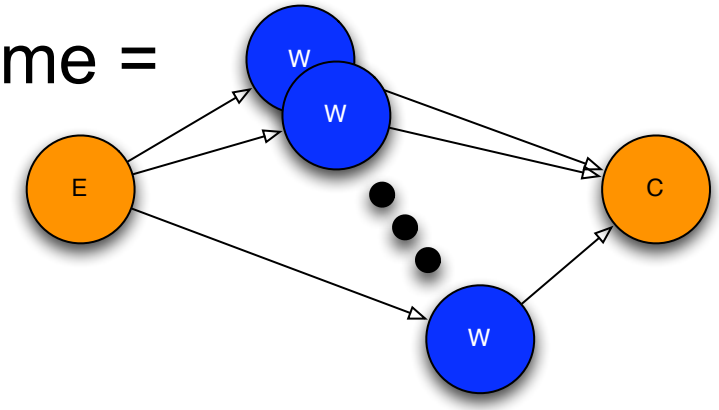
Skeleton systems: implementation templates

- Template
  - know, parametric, efficient process network implementing skeleton
  - performance model use to predict/monitor perf
- Compiler
  - generative process
  - heuristics



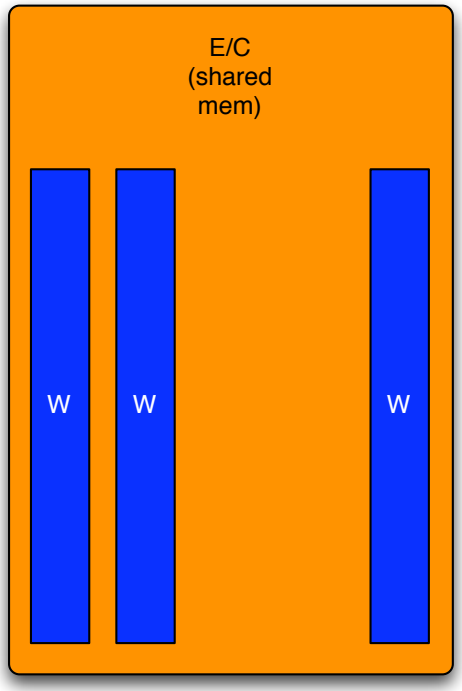
Sample template (1)

- Skeleton: Task Farm
- Target: COW/NOW
- Semantics: given x1 ... xn compute f(x1) ... f(xn) in parallel
- Implementation: Master worker (N workers)
- Performance model: Service time = max { Tcomm, Tseq/N }
- Process network: Emitter, string of Workers, Collector



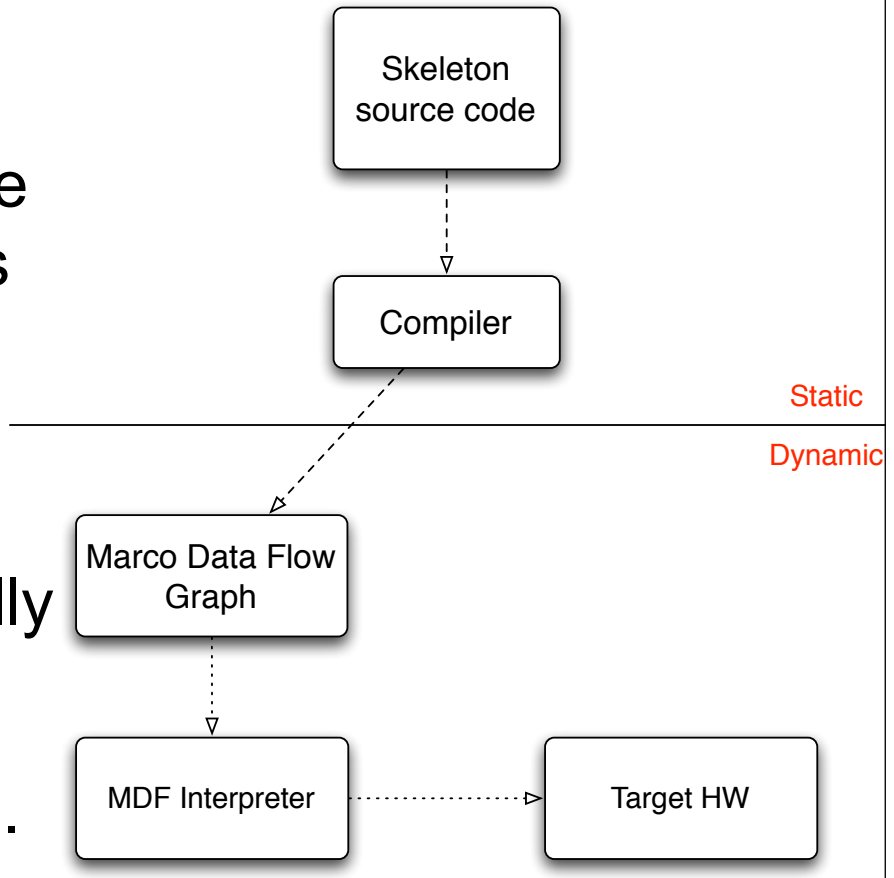
Sample template (2)

- Skeleton: Task Farm
- Target: Multicore/SMP
- Semantics: given x1 ... xn compute f(x1) ... f(xn) in parallel
- Implementation: Master worker (N threads)
- Performance model: Service time = max { Tcomm_shm, Tseq/N }
- Process network: One process, multithreaded



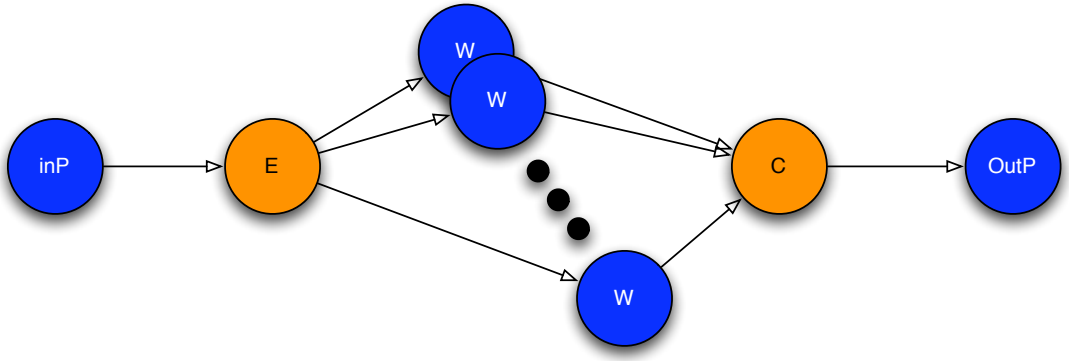
Skeleton systems: macro data flow

- Macro Data Flow
  - MDF instructions compute large sequential functions
- Distributed interpreter
  - fetch & execute fireable instructions from a logically centralized pool
  - optimization, heuristics, ...



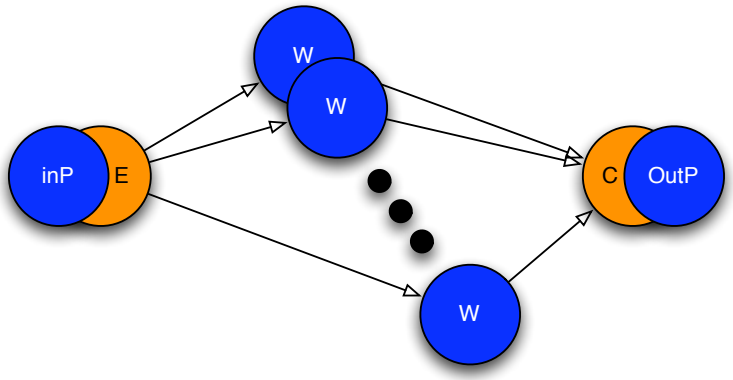
Sample application

- UC1
  - pipeline(seq,farm,seq)
  - plain template composition
  - number of workers estimated (perf. model) or dynamically optimized

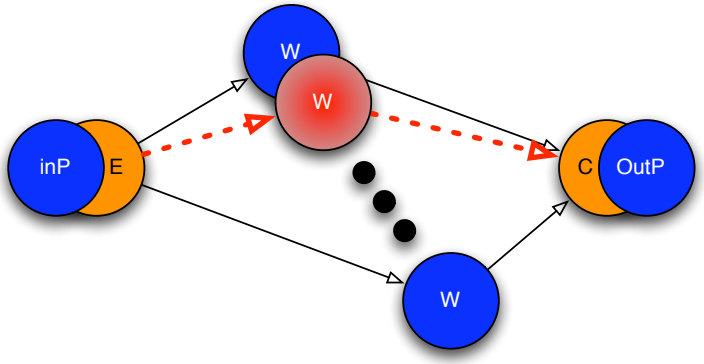


Sample application (2)

- Optimizations
  - grouping seq processes on different PEs



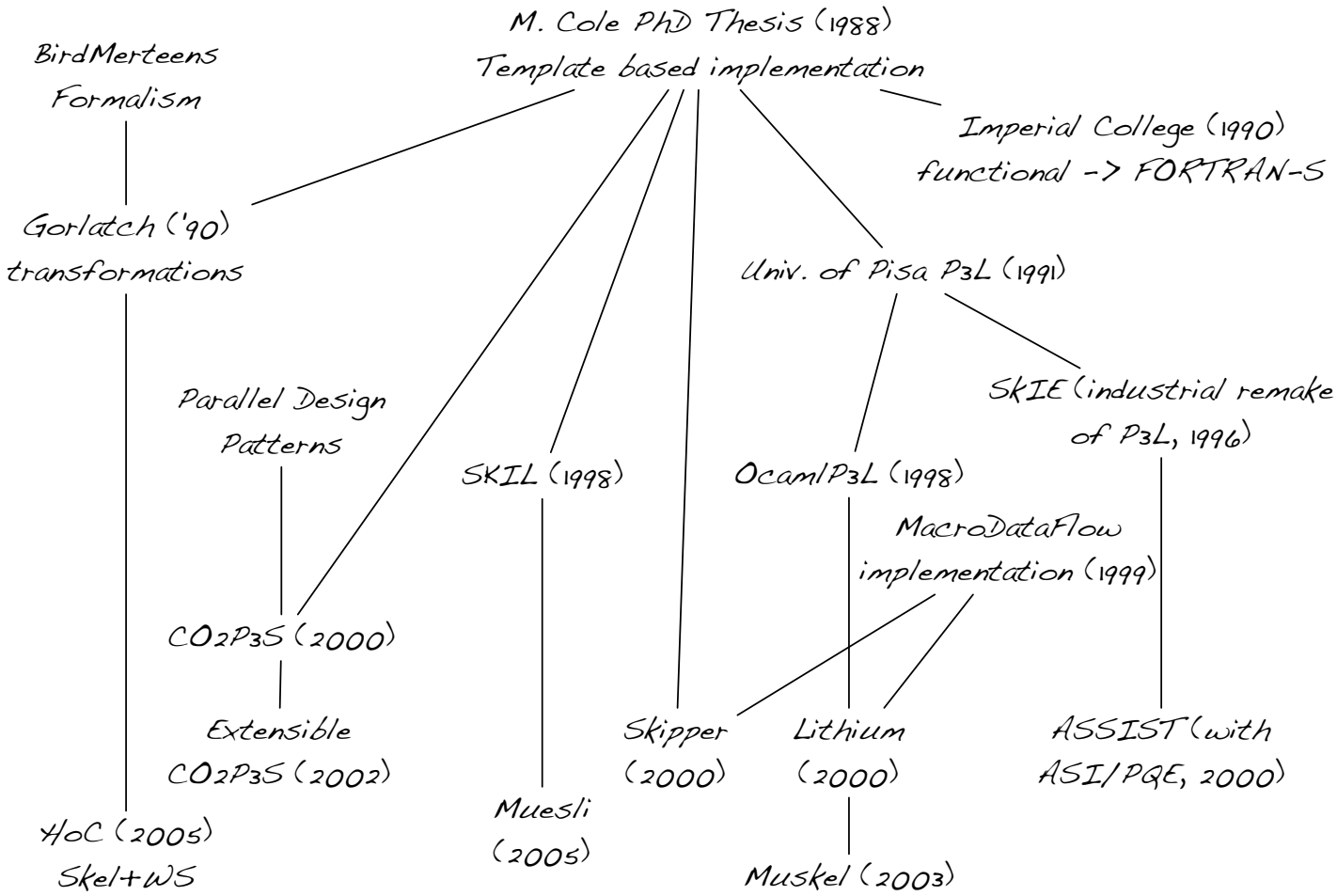
  - more complex opts: secure comms on nodes reached through public network segments



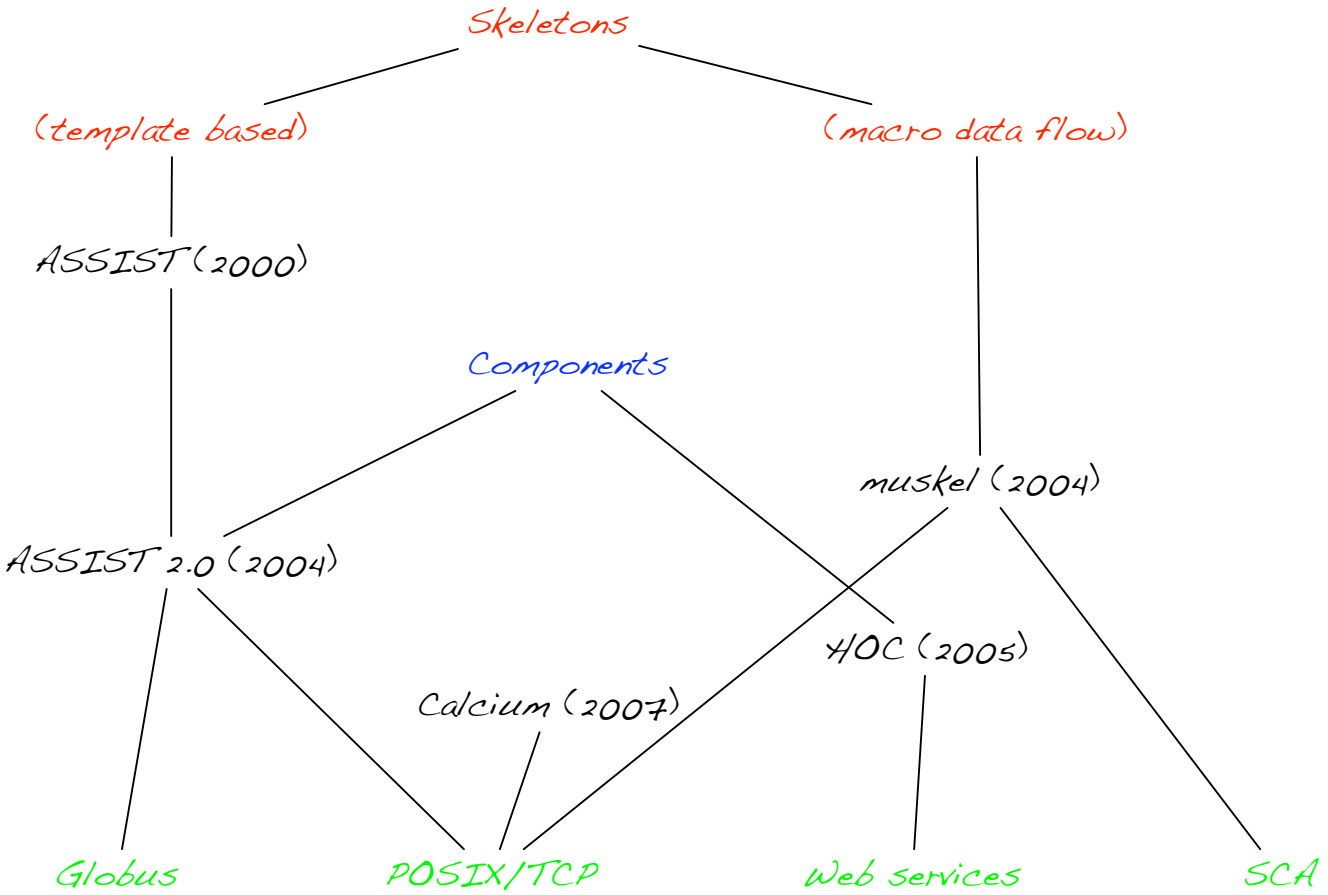
Results (expected)

- Raise level of abstraction (programmers)
- Hide target hw
- Implement state of the art solutions
  - without reinventing the wheel for each application
- Improve possibilities to implement autonomic management
  - knowledge from tools vs. from programmer code analysis

Structured parallel programming: the background (part of)



Structured programming: grid



Structured programming: grid

- Phase 1: plain skeletons (ASSIST ‘00, muskel ‘04)
  - POSIX/TCP or Globus
- Phase 2: components (ASSIST 2.0 ’04, HOC ‘05)
  - ... + WS
- Phase 3: autonomic components (ASSIST 2.0, GCM/ProActive)
  - ... + gLite, Unicore, ... , SCA

Skeletons to components

- mature component technology
- asymptote of the skeleton methodology
  - composite components model skeletons
  - skeleton parameter as components
    - management made easier (black box components)
- further separation of concerns
  - component framework: deployment, lifecycle, ...
  - composite components: other nonfunctional concerns

Components to autonomic

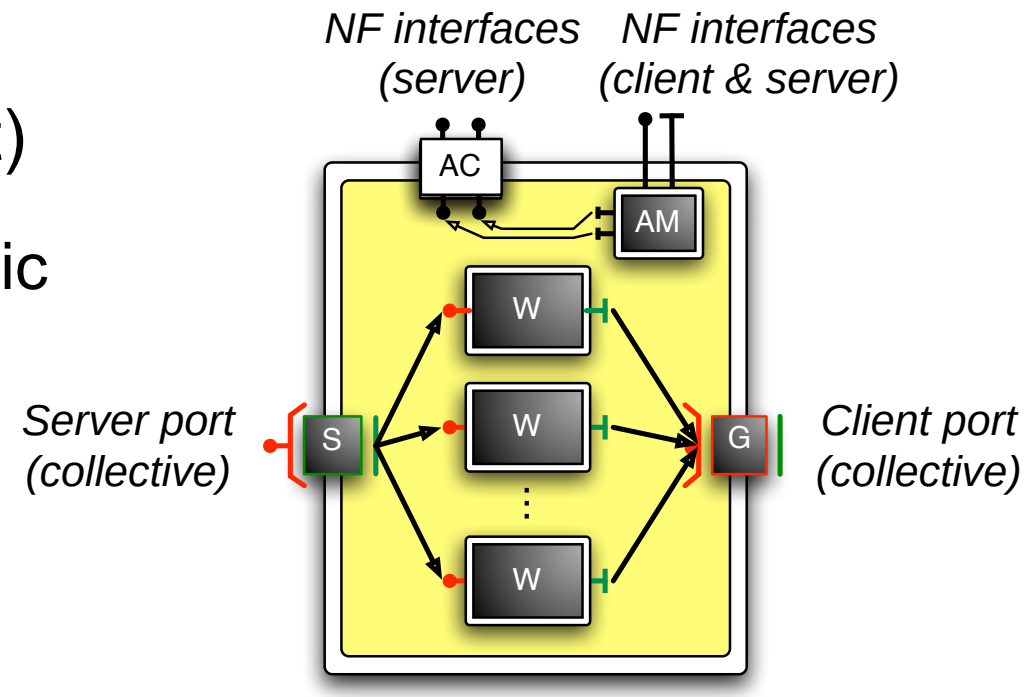
- Self-* features added
  - healing, configuration, protection, optimization
- Composite components
  - autonomic managers added (active)
- “Inner” components
  - autonomic controllers added (passive)

Behavioural skeletons (CoreGRID/GridCOMP)

- Exploit skeleton idea for management
  - Common parallel programming paradigms, management can be pre-determined (in a parametric way)
  - Capturing several aspects of management
    - Optimization, healing, configuration, protection
  - Active and passive management
    - hierarchical application composition

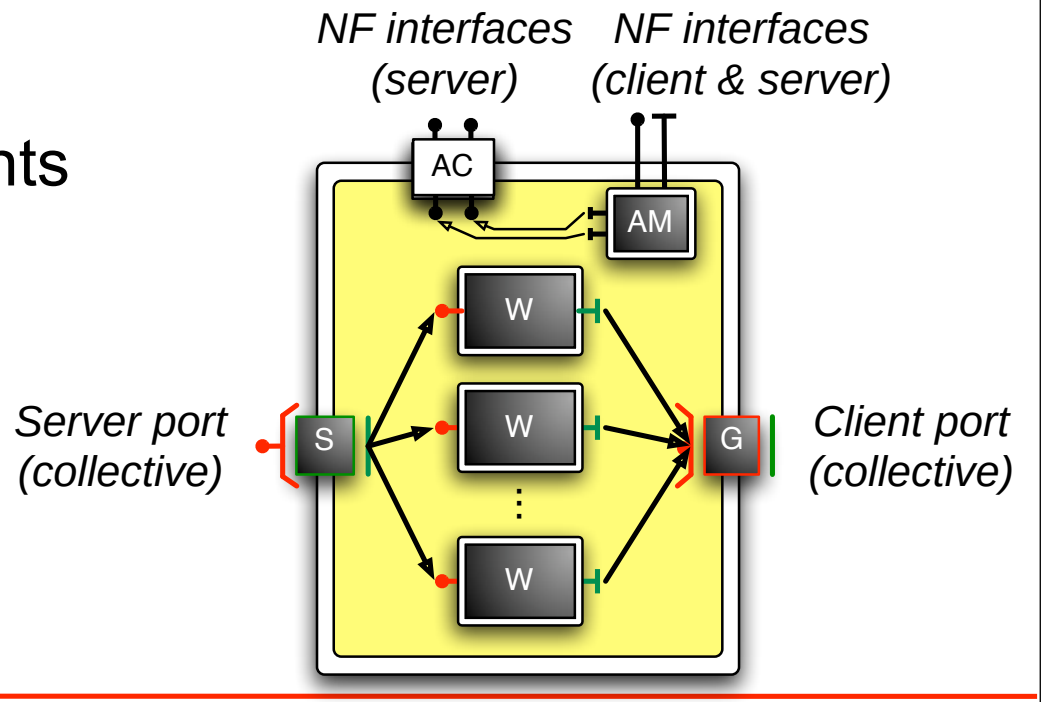
Sample behavioural skeleton

- Passive (AC is a controller)
  - component can be asked to behave in a specific way
- Active
  (AM is a component)
  - AM takes autonomic decisions concerning component behaviour



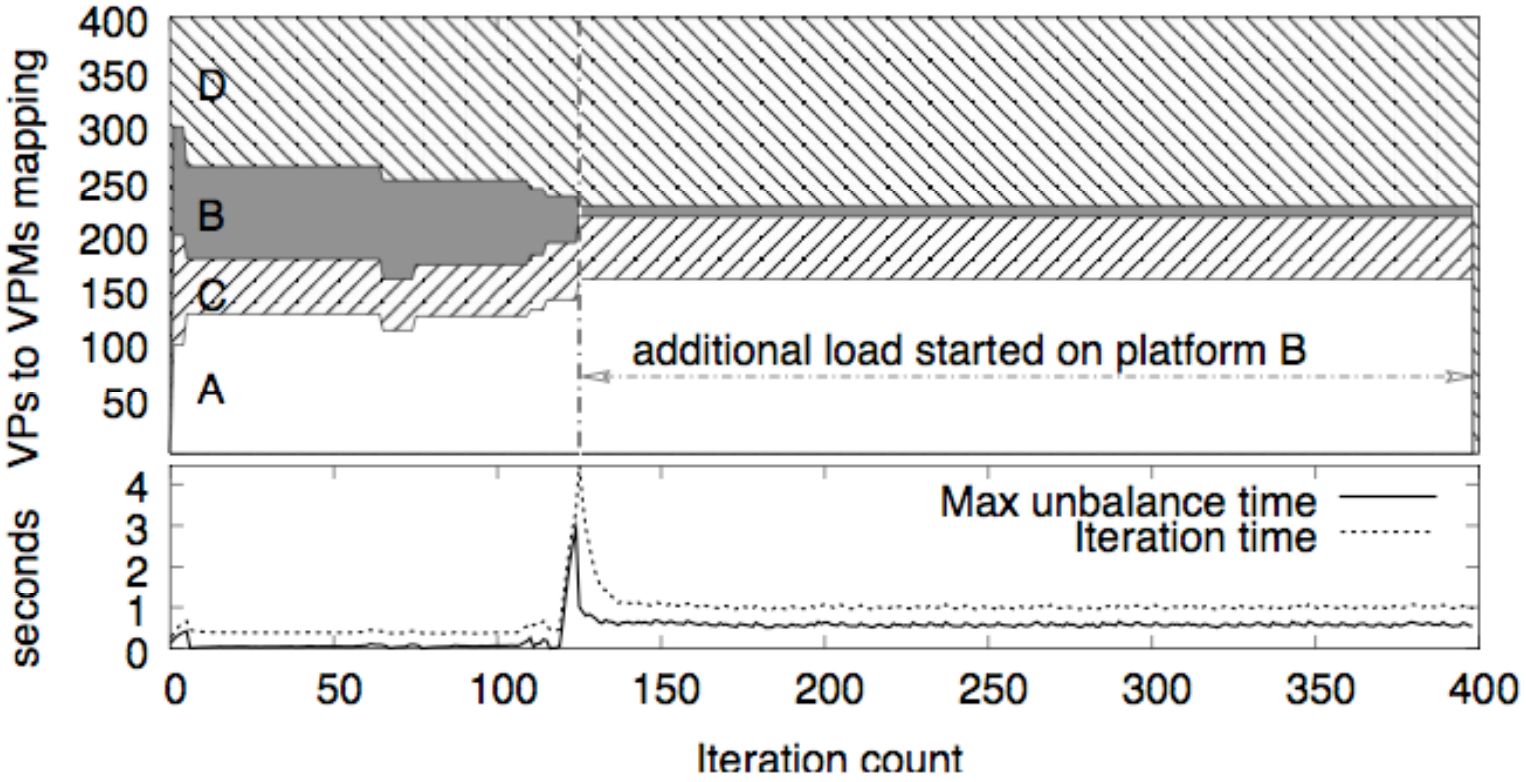
UC1

- S is a unicast
- G is a collect from any
- W compute F
  - these are components
- AM
  - fault tolerance
  - NW adaptation



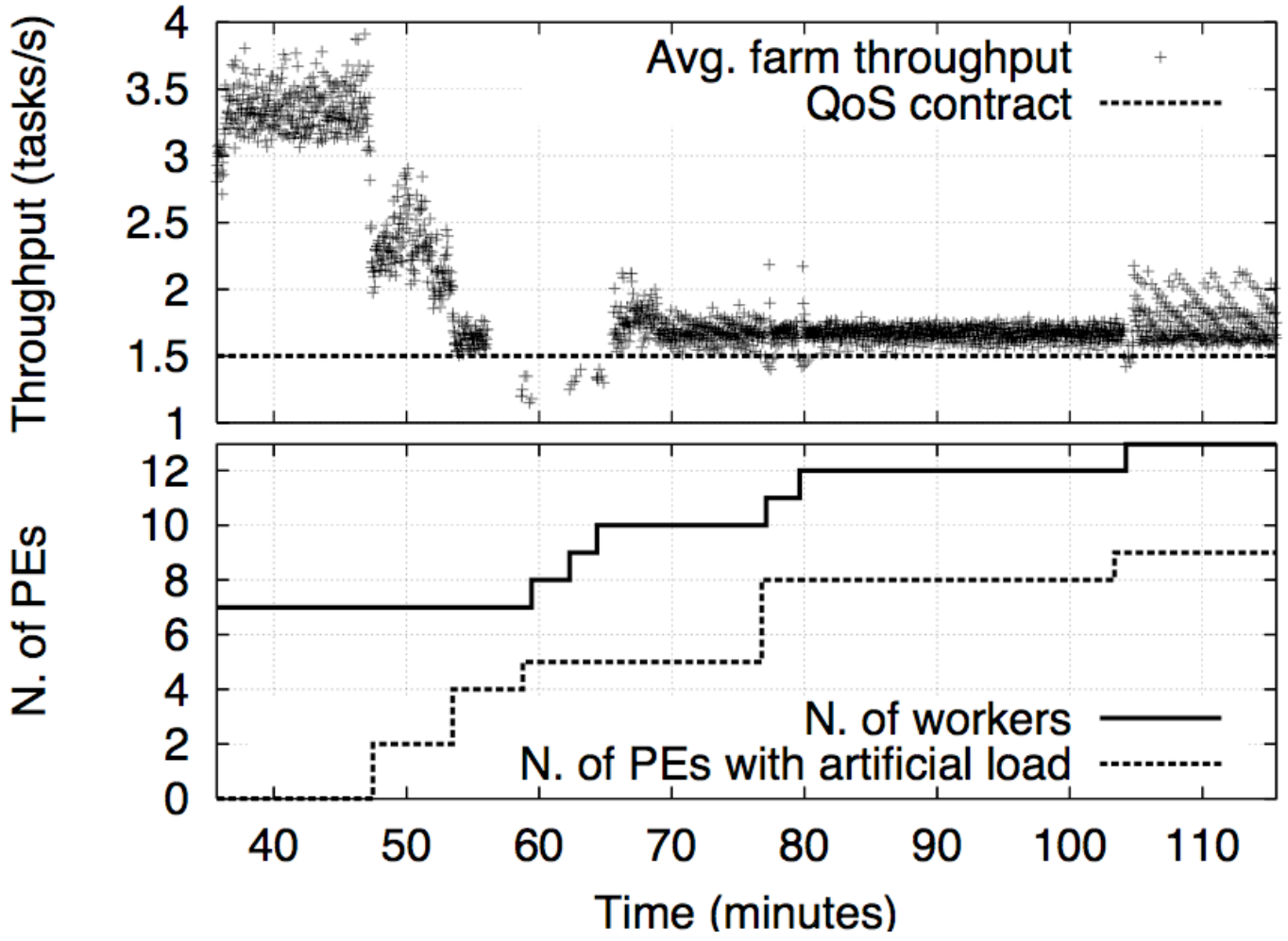
Which kind of results ?

- ASSIST 2.0



Which kind of results ? (2)

- GCM



Conclusions

- Structured parallel programming
  - demonstrated effective in HPC
  - successful proof of evidence on grids
  - supports autonomic control/management
- What's needed (still)
  - engineered environments
  - support for generic legacy components

Thank you for your attention

- any questions?
  - marcod@di.unipi.it