

# High Performance Software Components

Steven Newhouse

Anthony Mayer

John Darlington

High Performance Informatics Group

Department of Computing

Imperial College, London

{sjn5,aem3,jd}@doc.ic.ac.uk



Funded by EPSRC: GR/N13371 & PhD Studentship 979063079

## Outline

- Motivation & Background
- Software Abstractions & Components
- HP Software Component Architecture
- Linear Algebra Example
- Summary & Further Work

## Motivation

To bring the power of High Performance Computing facilities and applications transparently to the desktop

- HPC becoming ubiquitous in science, engineering, medicine and commerce
- Complex multi-disciplinary applications
- Emerging grid computing environment
- Use of HPC applications by ‘dumb’ users
  - Ease of access through integrated computation resources
  - Ease of programming through high level composable abstractions
  - Ease of comprehension through interactive visualisation

## Challenges from Computational Grids

- Computational platforms may not be known in advance
- Migration of long running jobs
- Heterogeneous distributed computational resources
- Complex security, access and usage policy
- Varying network bandwidth and latency
- Applications may need to be partitioned over multiple physically distributed computational resources

Need to extend current programming models and provide suitable computational environments

## Environment Wish List

- Build programs from high level definitions
  - Sophisticated Abstract Data Types
- Promote reuse and sharing of routines
  - Define interfaces / components
- Implementation flexibility
  - Understanding of an ADT's own behaviour to support optimal partitioning and map (eg. parallel, serial, tiled methods)

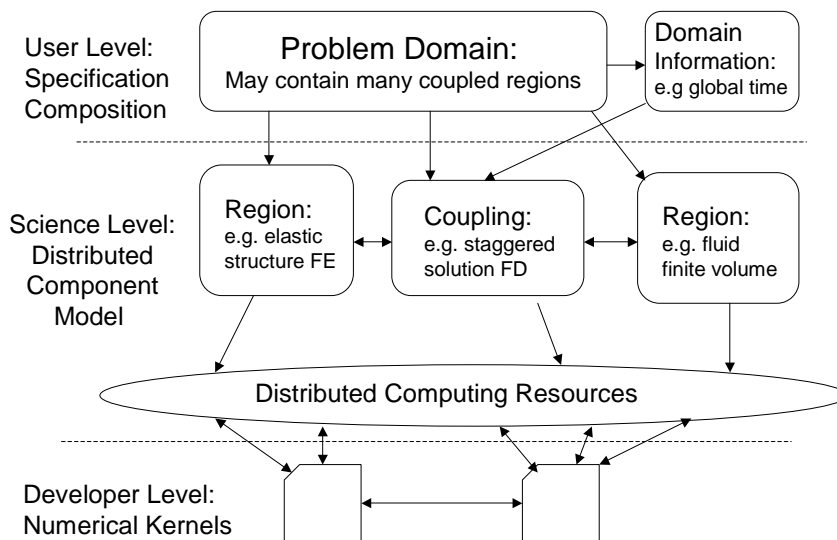
## Previous Work

- Skeletons (eg. P<sup>3</sup>L, SPF & SCL)
  - Optimise data layouts within a component and between components
  - Use performance models to guide component selection to find optimal overall performance
  - Use performance models to ensure optimum component performance on a computation resource
- Components (eg. CCA, Ligature, Cardiff)
  - Application level composition
  - Common Interface Specification

# Compositional Skeletons Approach

- Advantages
  - Demonstrated flexibility of compositional approach
  - Simple performance models supported effective compositional decisions
- Disadvantages
  - Combination of functional and F77/C languages
  - Poor take up by applied user community

## Abstraction Levels



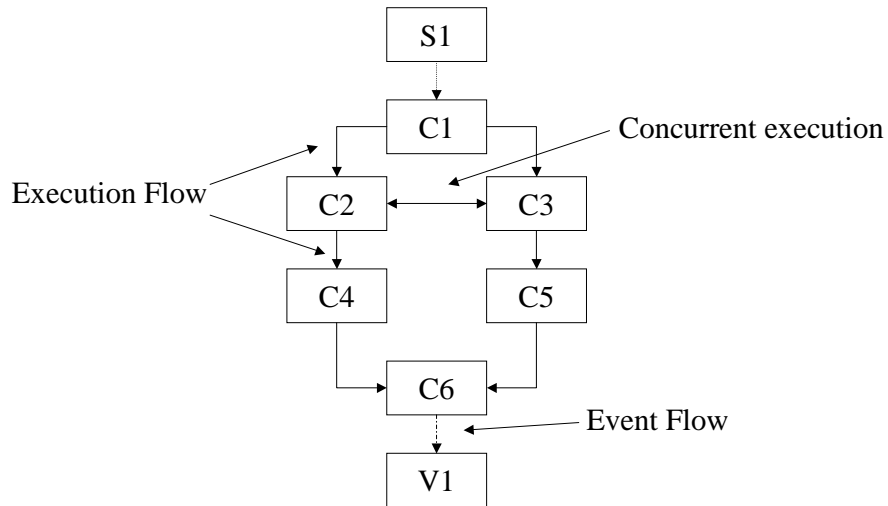
## Software Structure

Component Definition (XML encoded Metadata)
Coordination Layer (eg. Java Beans, RMI, CORBA, etc)
Component Implementation (Fortran, C, C++, Java etc)
Inter & Intra Component Communication (MPI, TCP/IP, CORBA, etc)

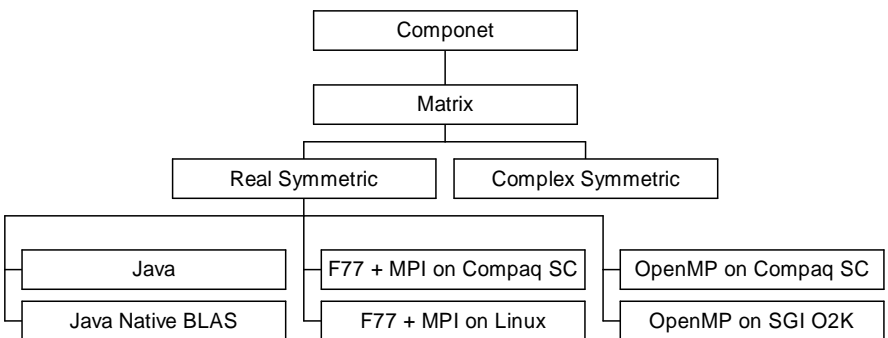
## XML Application Description

- User information (who, what, when & why)
- Component Network
  - Component Instance (id, name, source, properties)
  - Component Links (Type: event, concurrent, flow)
- Repository
  - Components (id, information, default properties, parent, ports, ACL)
    - Implementations (architecture, performance model, executable)

## The Component Network



## The Component Repository

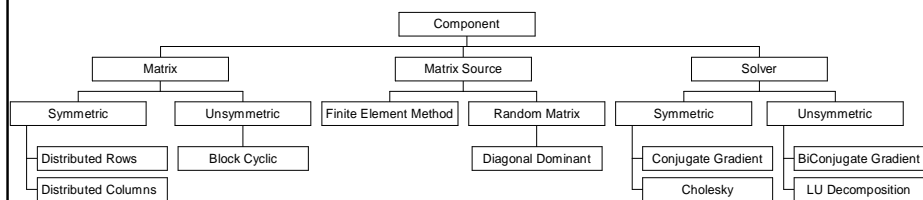


Implementations: Specified architecture or library requirements

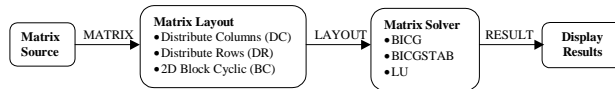
## Hybrid Skeletons & Component Model

- High-level abstractions essential for rapid adoption by user community
- Compositional approach allows flexibility & promotes reuse
- Clear separation between user intent and implementation
- Performance models and intelligent composition maintains program performance
- Separation between design and run-time configuration
- Component approach enables visual programming
- Fine application granularity presents opportunity for back end scheduling & optimisation

## Linear Algebra Components

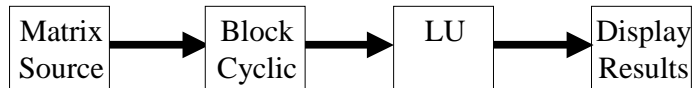


## Automatic component selection



1. Select your matrix source (derived from a matrix layout component)
2. Select a matrix solver and connect to the matrix source
3. Select a result handler (eg, screen, file etc.)
4. Execute

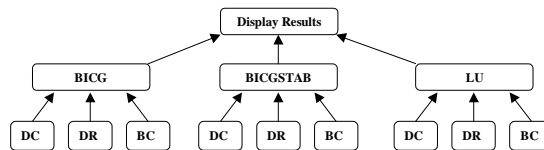
## Explicit component selection



- Supports 'smart' user
- Exploits performance models to find optimum partitioning
- Uses software repository as flexible toolkit

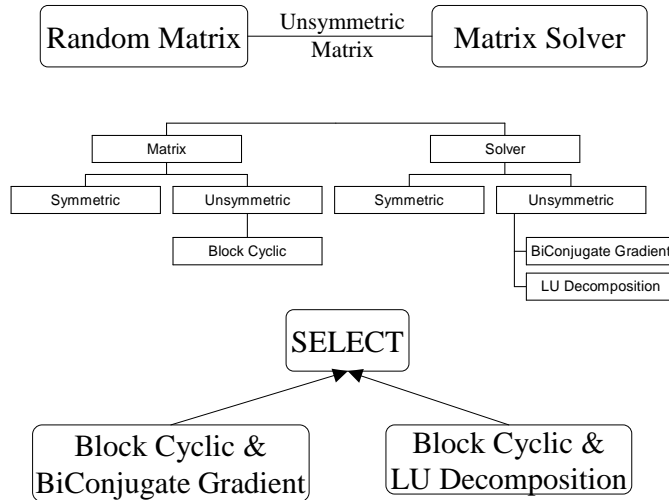
## Cross-component Optimisation

- Static Optimisation - Expand and examine correct implementation options

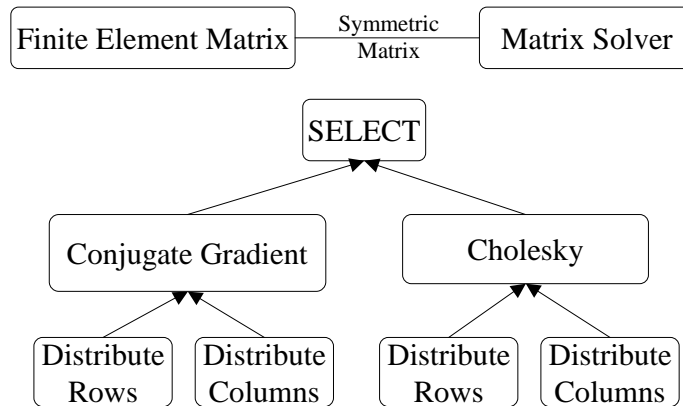


- Dynamic Optimisation - Run-time optimisation and lazy library techniques between components

## Static Component Optimisation 1



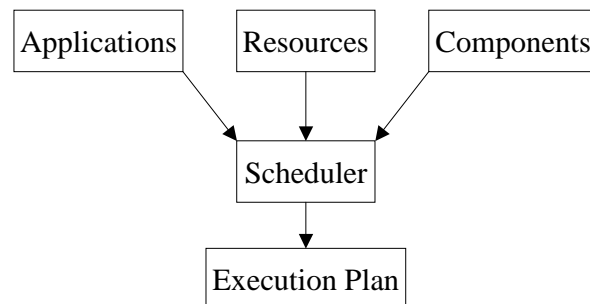
## Static Component Optimisation 2



## Dynamic Optimisation

- Delayed Communication
  - BSP: Buffer results until synchronisation point eg. Communicating matrix coefficients
  - DESO: Delayed Evaluation Self Optimisation - optimal data layout to reduce communication in parallel matrix algebra (Beckmann & Kelly)
- Merged Communication
  - Knowledge of distributed ADT's find optimal communication strategy eg. KeLP

## Multi-Application Level Scheduling



## Summary

- Component Repository allows code & component sharing
- Component hierarchy shows implementation options
- Enables the 'best' implementation to be selected 'intelligently' for a given application on any architecture
- The 'best' combination of methods and architectures can be selected from those that are or will be available

## Further Work

- Development of HPC parallel component applications in Java & native languages
- Exploit the composition framework for intelligent scheduling
- Deployment and use in local and national Computational Grid environments