Optimisation of Component-based Applications within a Grid Environment

Nathalie Furmento, Anthony Mayer, Stephen McGough,
Steven Newhouse, Tony Field, and John Darlington

Imperial College of Science, Technology and Medicine, Department of Computing,
Huxley Building, 180 Queen's Gate, London SW7 2BZ, UK
icpc-sw@doc.ic.ac.uk
http://www-icpc.doc.ic.ac.uk/components/

Abstract. Effective exploitation of computational grids can only be achieved when applications are fully integrated with the grid middleware and the underlying computational resources. Fundamental to this exploitation is information. Information about the structure and behaviour of the application, the capability of the computational and networking resources, and the availability and access to these resources by an individual, a group or an organisation.
This paper describes an integrated grid environment that is open, extensible and platform independent. We match a high-level application specification, defined as a network of components, to an optimal combination of the currently available component implementations within our grid environment. We demonstrate the effectiveness of this architecture through high-level specification and solution of a set of linear equations by automatic and optimal resource and implementation selection.

1 Introduction

The emergence of computational grids presents a number of challenges for the future of high performance computing. The grid, as discussed in [1], is defined as a wide-area network of heterogeneous computing resources, characterised by distinct administrative domains, and diverse operating systems and architectures. While it has been demonstrated that computational grids can provide a huge pool of potential high performance resources, the dynamic nature of the environment may inhibit efficient high performance computing; resources may become unavailable at any time, and it is impossible for the application scheduler to control the availability or dynamic properties of the resources unless it is closely coupled to a specific architecture.

SC2001 November 2001, Denver (c) 2001 ACM 1-58113-293-X/01/0011 $5.00

The computational efficiency of distributed parallel computing is strongly determined by network bandwidth and latency, both of which may vary over time in a grid environment. Thus the optimal mapping of the application code to the resources can only be achieved if the grid services and scheduling software are always aware of the application's requirements. Such actions must also take place within the constraints of locally defined security and usage policies.

In this paper we demonstrate an extensible component framework which separates implementations from abstractions, and maintains those abstractions at run-time.

This paper makes the following contributions:

- the definition of CXML (an XML schema) which is used to describe component meta-data and act as a common intermediate language between the elements of the component framework.
- the separation of component interfaces from their implementations through abstraction and encapsulation. This enables component portability by enabling multiple architecture specific implementations for a given interface.
- the declaration of component behaviour to determine inter-component control flow dependencies.
- the run-time selection of the best combination of implementations through composite performance modelling, using the performance and behavioural component meta-data.
- demonstration of implementation selection for two different user specified optimisation goals, minimising execution time, and minimising execution cost.

2 Background

2.1 Abstraction within Scientific Computing

When an applied computational scientist develops a program to solve a particular problem, they make use of domain-specific knowledge in the construction of the application [2]. For example, when building a finite element model to determine cable deflection it is known that the resulting stiffness matrices are symmetrical and tri-diagonal. This knowledge is used in the act of programming, such as in the choice of linear solvers and matrix storage patterns. However once the program is written the abstraction of this knowledge is lost and only the low-level (typically C++ or FORTRAN) code remains.

Where the programming language utilises high level abstractions as programming constructs (such as in object-oriented programming and functional languages [3–5]), this knowledge may be retained and used in compile-time optimisation, but once again may be lost in the translation to low-level high performance executables.

In addition to the end-user's knowledge regarding the application, a developer's knowledge of the performance characteristics and behaviour of a low-level

library is also lost once it has been linked into an application. Though dynamically linked libraries are not bound until run-time, there is usually little or no meta-data that describes such libraries that would enable intelligent scheduling, especially within the context of the computational grid.

Retaining this high-level knowledge (application behaviour, properties, and low-level performance characteristics) is essential to optimising the implementation at run-time [6].

2.2 Component Architectures

Component based design is a well established software engineering pattern. Widely used general purpose component systems include JavaBeans [7], which is based upon the Java language, CORBA [8], which provides language interoperability and Microsoft's DCOM [9]. These systems provide software bindings between components, but do not maintain any form of high-level information or meta-data.

There are also practical examples of component based systems in use within high performance computing, including the Linear System Analyser [10], the Common Component Architecture [11] and Ligature [12]. These systems provide additional compositional meta-data beyond the basic software bindings provided by CORBA, DCOM etc. However they do not exploit performance data to guide component optimisation, and implementation selection is not developed fully.

Component based design is intended to replicate the assembly line process of mechanical engineering within software engineering, by providing methods of encapsulation and abstraction that enable separate development of an individual component without causing disruption to those components that are dependent upon it. Encapsulation is ensured through requiring all context dependencies to be made explicit in the component interface, which is also an abstraction of the component.

2.3 Components within a Grid Environment

In order to exploit the high-level information for scheduling in a grid context it is necessary to adapt the component based design pattern to provide the portability, mobility and high performance while retaining the facility to maintain and export the required component meta-data at run-time.

The separation between the high-level abstraction and low-level implementation allows multiple implementations to exist for a single abstraction. Within the conventional component-based design paradigm, this feature is used to provide robust incremental software development over time, by allowing distinct implementations of a given interface. This separation may also be used to enable portability of the high-level abstraction, by providing architecture specific implementations of a machine independent abstraction. In the context of a computational grid, the availability of resources may change during run-time, hence efficient deployment within a dynamic resource environment may necessitate changing the implementation of high level abstraction during execution.

To support the information gathering and deployment phases of application execution between different resources and organisations, we are developing a grid middleware that consists of distinct private and public areas. This grid architecture operates at a comparable level of abstraction to the Legion system [13]. An implementation of this architecture, using Java and Jini, is being tested over our local resources and is being used to support this work. Details of this architecture may be found elsewhere [14, 15].

3 An Extensible Component Framework

We have defined a distributed environment for component based design and deployment that makes use of component meta-data and the separation between interface and implementation. As such it is specifically designed for use within a dynamic heterogeneous resource environment where many correct implementations may exist for a single component interface.

This framework consists of a number of distinct tools, which are related through the common use of an intermediate language. This is the Component - eXtensible Markup Language, or CXML, which is a realisation of the XML meta-language [16]. This language provides a means by which the diverse units of the framework may communicate, and is used to describe the meta-data for the abstract components, the component implementations, resources and applications.

The framework is outlined in Figure 1. The tools are geared towards distinct user roles in the design cycle; there is a separation between the end-user, who wishes to utilise existing component abstractions to complete a problem solution, and the developer who wishes to provide components and component meta-data for end-user use. The design and deployment cycle for an application in the framework is straightforward.

1. **Component specification** When a new component is developed, its component specification (a CXML document) is placed in the component *repository*, together with meta-data that describes its behaviour and interface.
2. **Component Implementation** All components must have at least one implementation. Each component implementation corresponds to a particular component specification. New implementations are placed within the repository, along with meta-data describing their performance characteristics and resource requirements. The implementation meta-data is a CXML document, distinct from but linked to the component specification.
3. **Problem Definition** A new application is created by composing instances of the abstract components within the repository. The possible compositions are indicated by the component meta-data. The end-user may use tools such as a problem solving environment to produce the *application description document*, which is a CXML specification of the complete component composition.
4. **Run-Time Representation** At deployment, the application description document is converted to an active Java representation of the application, by

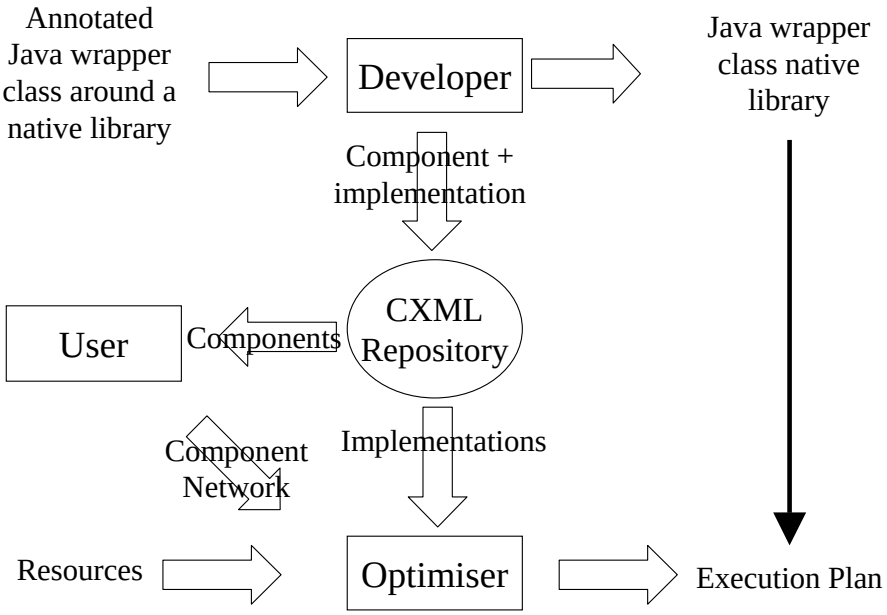

Fig. 1. Extensible Component Framework

utilising the component specification meta-data within the repository. This run-time representation encodes all the meta-data for the component application, and is maintained throughout the deployment of the application allowing further run-time optimisations to take place.
The run-time representation is described more fully in [17], where it is demonstrated that for a simple example the run-time overheads of such a mechanism are within acceptable bounds.

5. **Implementation Selection** The *application mapper* uses the run-time representation, meta-data describing the available resources, provided by *grid information services*, and implementation meta-data found in the repository to decide which of the available resource and implementation combinations are to be chosen for the application's components. This is encoded in an *execution plan*, which is passed to the *grid deployment services* for execution. The application mapper uses composite performance modelling, as proscribed by the execution behaviour encoded in the component meta-data to enable its decision making process.
6. **Run-time optimisation** During execution the grid resources may change, due to unforeseen circumstances, administrative actions, network issues, etc. At any point the grid services may return to the *application mapper*, which

can then recreate a new execution plan utilising the latest resource information.

All of these tools are essentially independent; though they all use CXML as a target language, and utilise CXML specified information, they may be developed independently. The component framework itself may be thought of as a component based system, with CXML as a unified but extensible interface. The grid information and deployment services do not depend upon the potential structures of the component application systems, nor are the component applications dependent upon the format of the grid services.

A prototype environment has been produced to demonstrate the design pattern described above, providing a visual environment that allows access to repository meta-data, application composition and resource acquisition through grid services. This is described fully elsewhere [18].

4 Meta-Data for Run-Time Optimisation

Throughout this section we examine a simple example application to demonstrate the various features of the framework.

Example: Linear Equation Solver. A canonical example of a high performance computing problem is the solution of systems of linear equations in parallel. Such computation regularly occurs in a wide range of scientific computing problems. When this application is represented as a component system, it consists of three component instances; a simple linear equation generator, a solver, and a simple display component. These are connected as shown in Figure 2. While it is highly unlikely that such simple problems as the solution of linear systems would use the computational grid, components such as these are likely to form the basis of larger more complex scientific problems (as described in section 7).

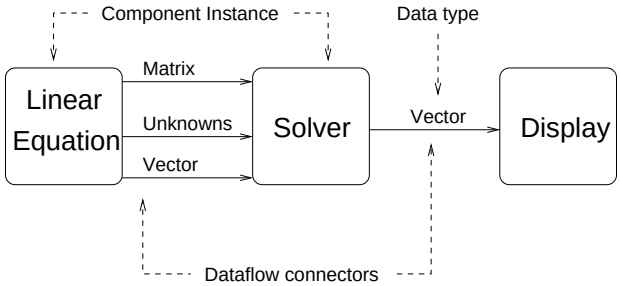

Fig. 2. Linear Equation Solver

4.1 Interface Meta-Data

The repository contains meta-data that describes the interfaces for component types. A component type may have methods which are grouped into ports, and public data values, which are accessed via these ports. *Outports* are typed collections of available methods, and *inports* are typed references to the outports of other components. This port mechanism allows the composition of components. Data values are represented by *data* elements. The CXML meta-data describing the component interface for the linear equationsolver shown in Figure 2 is indicated in Figure 3 below:

```
<component package="icpc.LinearEquationSolver"
   name="LinearSolverRowsColumnsUnsymmetric" version="1">
     <parent package="icpc.framework.core" name="Component" />
     <inPort objectPackage="icpc.framework.core" objectName="Integer"
       portName="unknowns" />
     <inPort objectPackage="icpc.Matrix" objectName="DgeRCj"
       portName="matrix"/>
     <inPort objectPackage="icpc.Matrix" objectName="RealVector"
        portName="vector"/>
     <data exposure="public" objectPackage="icpc.Matrix"
         objectName="RealVector" dataName="solution" >
       <outPort>
        ...
       </outPort>
     </data>
</component>
```

Fig. 3. CXML fragment: Linear Equation Solver Component

4.2 Application Descriptions

As a component may have state, a realisation or instance of a component type is distinct from the abstract component type itself, in an analogous fashion to the distinction between object and class in object oriented design.

An application description consists of a composition of component *instances*. This composition is based upon the port mechanism, which encapsulates method calls and data access between components. The ports are linked by *connectors* that represent dataflow between the components.

The application description document for the example shown above is given in Figure 4.

4.3 Behaviour Specification

Implementation selection requires analysis of the behaviour of the composite application, for while a performance model may be supplied by a developer as

```
<application>
  <network>
    <instance componentName="Source"
         componentPackage="icpc.LinearSource" id="1">
              <property name="degrees of freedom" value="100"/>
    </instance>
    <instance componentName="Solver"
              componentPackage="icpc.LinearSolver" id="2"/>
    <instance componentName="DisplayVector"
              componentPackage="icpc.Matrix" id="3"/>
    <dataflow sinkComponent="2" sinkPort="matrix" sourceComponent="1"
              sourcePort="matrix"/>
      ...
  </network>
</application>
```

Fig. 4. CXML Application Description fragment

implementation meta-data, the composite performance of the complete application depends upon the relationship between the components of the application. Component context dependencies must be explicit by definition [19], and as such must be specified within the high-level abstraction.

Such analysis does not require a detailed programming language semantics that expresses the complete behaviour of the component's implementation, but must indicate where a method call on a given component results in subsequent calls to other components, and in what fashion such calls are made. Behavioural specifications are implementation independent, as by definition any implementation that satisfies the interface must have the same external dependencies.

Thus the CXML component meta-data contains a specification of the inter-component behaviour of all method calls, but does not describe that behaviour which is internal to a given component. Behaviour is indicated for any method, and is hence attached to the component specifications for its outports. The behavioural specification is a recursive structure, built from `<block>` elements. The leaves of this structure are either `<call>`s, which indicate a method of the implementation is to be called, or `<get>` and `<set>` tags which indicate access to data elements which may be on other components (accessed via the port mechanism). The behavioural CXML also includes elementary control structures such as conditional and iteration elements.

In the linear equation solver example the source component possesses a Boolean `<data>` element which acts as a flag to indicate whether the matrix and vector have been `generated`. When a call arrives requesting either the matrix or vector, this flag is checked. If the data has been cached, it is supplied directly. Where it hasn't yet been generated, both matrix and vector are created simultaneously (as they represent left and right hand sides of the linear system their creation is a single process). This CXML behaviour is expressed in Figure 5.

```
    <component package="icpc.LinearEquationSource"
      name="LinearEquationSourceRowsColumnsUnsymmetric" version="1">
      <parent package="icpc.framework.core" name="Component" />

      <data exposure="private" objectPackage="icpc.framework.core"
         objectName="Boolean" dataName="generated">
        <default>FALSE</default>
      </data>

      <data exposure="property" objectPackage="icpc.framework.core"
       objectName="Integer" dataName="unknowns">
        <default>10</default>
      </data>

  ...

      <data exposure="public" objectPackage="icpc.Matrix"
        objectName="DgeRCj" dataName="matrix">
       <outPort>
        <behaviour method="getMatrix" >
         <block execute="sequential" >
          <conditional dataName="generated" >
           <false>
            <block execute="sequential" >
             <call portName="equations" method="create"/>
             <set dataName="generated" >TRUE</set>
            </block>
           </false>
          </conditional>
          <return/>
         </block>
        </behaviour>
       </outPort>
      </data>

   ...

</component>
```

Fig. 5. CXML: Linear Equation Source Component

This demonstrates that the relationship between a component's behaviour and the meta-data may change at run-time. In this example the altering of the meta-datum `generated` is trivial, but changes the way in which the matrix and vector are accessed, which may be non-trivial for larger problem sizes within a grid environment.

4.4 Implementation Selection and Component Parameterisation

The run-time representation consists of a network of Java objects representing the various elements of the application description and component meta-data [17].

The application mapper has a number of possible ways of optimising the component deployment. The first of these is implementation selection. A given component's run-time representation may accept any number of available implementations at run-time, and it is the role of the application mapper to choose the appropriate implementation for the available resource.

The application mapper selects an implementation for each component on the basis of the composite performance model of the application. As such it utilises the behavioural information to create a directed acyclic graph of the application's method calls, and from this compare the different possibilities for implementing the run-time representation. The recursive nature of the behaviour specification allows the composition of a performance model from atomic performance models of implementation method calls, provided as implementation meta-data and stored within the repository. These are assembled into a tree structure which representing the application's method calls (see Figure 6). To produce a composite performance model the tree is traversed by the application mapper, and those nodes of the tree that represent implementation method calls (represented by ellipses) contribute their performance model to the composite total. Different behavioural elements (iteration, concurrency etc.) allow different ways of composing these atomic models.

Where an application is distributed across multiple platforms, it is possible to utilise the application description and attendant component meta-data describing the application's dependencies to evaluate data transfer costs and schedule the deployment effectively.

While the application mapper may be extended to utilise any heuristic to perform the implementation selection, it currently provides a deterministic model, and requires empirical performance models of the implementation methods. We intend to use the principles developed in the AppLeS project to select implementations with complex parameterised performance models [20, 21], and to incorporate stochastic network information.

As the run-time representation is maintained alongside the executing implementations, it is possible for the application mapper to re-evaluate the implementation selection at any time during the application's execution. The run-time representation enables persistent data to be maintained between implementations [17]. In addition the application mapper may parameterise meta-data at run-time according to the resource. For example the block-size of a parallel matrix implementation, or the time step size of a finite difference problem may be customised when run-time information becomes available. This leads to incremental parameterisation and scheduling.

5 Implementation Selection without Diagonal Dominance

The linear equation problem (introduced in Section 4) requires the selection of an implementation for the source and solver components. The current application mapper builds a composite performance model for every possible combination of implementations by examining the inter-component control flow dependency encapsulated within the CXML. The source component generates a random unsymmetric set of real linear equations, for a given number of degrees of freedom, through either a C or Java implementation. The solver component has a wide choice of implementations, some of which are listed in Table 1. The performance models for the generation of the linear equations and the solution of those systems of equations are published in the accompanying technical report [22].

System	Processor	Processors	Language	Solution
PC (Linux)	AMD 900Mhz	1	Java	LU
				BCG
			C	LU
				BCG
			LAPACK	LU
Alpha	Alpha 667Mhz	1,4,9	ScaLAPACK	LU
				BCG
AP3000	UltraSPARC 300Mhz	4,9,16	ScaLAPACK	LU
				BCG

Table 1. Available solver implementations

In the case of the biconjugate gradient method, an empirical performance model may be supplied for a single iteration. The number of iterations required for convergence must be estimated. This estimate is necessarily pessimistic, with n iterations for a problem with n degrees of freedom. In addition to the cost per iteration is a small overhead for start up.

6 Implementation Selection with Diagonal Dominance

In many practical problems the matrices representing linear systems are diagonally dominant, that is the diagonal terms are significantly larger than the sum of the other terms within the matrix's rows or columns. The awareness of diagonal dominance is end-user knowledge, and may be included as meta-data within the application and repository information in order to enable further optimisation.

For example, if the linear equation source is replaced with a specifically 'diagonally dominant' source, the example application is altered to that in Figure 7.

This is represented in the CXML with the addition of the meta-data within the component description, firstly a *data* object within the linear equation source component:

```
<data exposure="public" objectPackage="icpc.framework.core"
          objectName="Boolean" dataName="diagonalDominance"/>
```

and as an *inport* within the solver component:

```
<inPort objectPackage="icpc.framework.core" objectName="Boolean"
        portName="diagonalDominance" />
```

The performance characteristics of an iterative solver may be known with greater reliability when the source produces a diagonally dominant matrix; the number of iterations a biconjugate gradient algorithm takes to converge may be known with greater certainty than in the case of a matrix where the diagonal dominance is unknown. Our augmented linear equation source produces matrices with diagonal terms that are increased in proportion to the size of the matrix. In general we find that the biconjugate gradient method requires $\sqrt{n}$ iterations to converge when operating on these matrices.

This knowledge may be used to provide distinct performance models for cases where the source linear system is diagonally dominant. These are provided within the accompanying technical report [22].

Figure 8 shows the predicted overall execution time for the composite linear equation system as the problem size increases. The application mapper selects the *best* combination of implementations to yield either minimum execution time (the solid line) or minimum resource cost. The shortest overall execution time is found by selecting the LU solver over 9 Alpha processors for small problems and then switching to the biconjugate gradient method for larger problems. When the selection criteria is based around the cost of the computational resources then the 1, 4 and 9 Alpha processors with the biconjugate gradient method are used in turn as the problem size increases. The current composite performance model does not include the data transfer cost between implementations on different platforms.

The computational cost resulting from each selection decision is shown in Figure 9. The total cost of each possible component implementation is evaluated from the estimated execution time together with the processor count. The cost per processor per unit time is defined as being equal for all the computational resources. The cost model forces a balanced approach to optimal resource selection with additional computational resources being used as the problem size increases.

In Figures 8 and 9 different symbols are used to represent the different implementation selections. The linear equations are always generated in either a Linux Java or C implementation. The Alpha cluster, the fastest available computational resource, is always used to solve the resulting linear equations.

7 Conclusions & Further Work

We have demonstrated that the retention and use of high-level knowledge at run-time allows implementation selection to be guided by the meta-data, and as such provides opportunities for policy based optimisation to minimise execution time or resource cost. A framework that utilises component based design to encode that meta-data and optimise via implementation selection provides high performance, portability and mobility in a dynamic resource environment.

Further work includes the development of the framework and its subsidiary mechanisms along the following lines:

Data Movement Costing The incorporation of data transfer costs within the cost model will allow a better measure of performance across a distributed environment such as the Grid. As it has been demonstrated that the framework will accept multiple costing schema, the addition of data movement costs will not require a major alteration of the framework.

Dynamic Optimisation The framework's extensible structure allows for the addition of plug-in heuristics to replace the simple 'examine all possible configurations'.

Run Time Instrumentation The dynamic optimisation of the application will require incremental knowledge of performance acheived and resource conditions. Hence it is intended that the framework be built upon grid instrumentation tools, such as the network weather system [23] and the tools developed as part of the GrADS project [24].

Richer Application Examples Work in progress includes the development of component annotations of finite difference and finite element code, applicable to a wide range of scientific disciplines.

Integration into federated Grid community As discussed in [18] and [15], this work is designed to be integrated with a system of application scheduling within a federated computational community, to allow the implementation selection and parameterisation to be combined with resource brokering and negotiation.

Such extensions to the framework will enable the benefits of component-based design to be delivered to prospective Grid users and applications.

References

1. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, July 1998.
2. Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End-User Computing*. MIT Press, 1993.
3. J. Darlington *et al.* Parallel Programming using Skeleton Functions. In *Lecture Notes in Computer Science*, volume 694, pages 146–160.
4. J. Darlington, M. Ghanem, Y. Guo, and H. W. To. Guided Resource Organisation in Heterogeneous Parallel Computing. *Journal of High Performance Computing*, 4(10):13–23, 1997.

5. P. Au, J. Darlington, M. M. Ghanem, and Y. Guo. Co-ordinating Heterogeneous Parallel Computation. In L. Boug, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par '96*, pages 601–614, August 1996.
6. Calvin Lin and S. Guyer. Optimizing the use of high performance libraries. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing*, August 2000.
7. JavaSoft. JavaBeans. `http://java.sun.com/beans`, October 1996.
8. Object Management Group. The common object request broker: Architecture and specification. formal document 97-02-25, July 1995. `http://www.omg.org`.
9. D. Chappel. *Understanding ActiveX and OLE - A guide for Developers and Managers*. Microsoft Press, 1996.
10. The linear system analyzer. `http://www.extreme.indiana.edu/pseware/LSA/LSAhome.html`.
11. R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *In Proceedings of the 8th High Performance Distributed Computing (HPDC'99)*, 1999.
12. Katarzyna Keahey, Peter Beckman, and James Ahrens. Ligature: Component architecture for high performance applications. *The International Journal of High Performance Computing Applications*, 14(4):347–356, Winter 2000.
13. A. S. Grimshaw and W. A. Wulf *et.al*. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40:39–45, 1997.
14. S. Newhouse and J. Darlington. Computational Communities: A Marketplace for Federated Resources. HCPN 2001, June 2000.
15. N. Furmento, A. Mayer, S. McGough, S. Newhouse, and J. Darlington. Building Computational Communities for Federated Resources. Accepted for Euro-Par 2001.
16. W3 Consortium. XML: eXtensible Markup Language. `http://www.w3c.org/XML`.
17. N. Furmento, A. Mayer, S. McGough, S. Newhouse, and J. Darlington. A Component Framework for HPC Applications. Accepted for Euro-Par 2001.
18. Nathalie Furmento, Anthony Mayer, Stephen McGough, Steven Newhouse, Tony Field, and John Darlington. An Integrated Grid Environment for Component Applications. Accepted for Grid2001.
19. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.
20. F. Berman and B. Wolski. The AppLeS project: A status report. In *Proc. NEC Symp. On Metacomputing*, 1997.
21. H. Dail, G. Obertelli, F. Berman, R. Wolski, and A. Grimshaw. Application-Aware Scheduling of a Magnetohydrodynamics Application in the Legion Metasystem. In *Proceedings of the 9th Heterogeneous Computing Workshop*, May 2000.
22. N. Furmento, A. Mayer, S. McGough, S. Newhouse, and J. Darlington. Performance Models for Linear Solvers within a Component Framework. Technical report, ICPC, 2001.
23. R. Wolski. Dynamically forecasting network performance to support dynamic scheduling using the network weather service, 1997.
24. F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, L. Gannon, K. Kennedy, C. Kesselman, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software support for high-level grid application development, 2000.

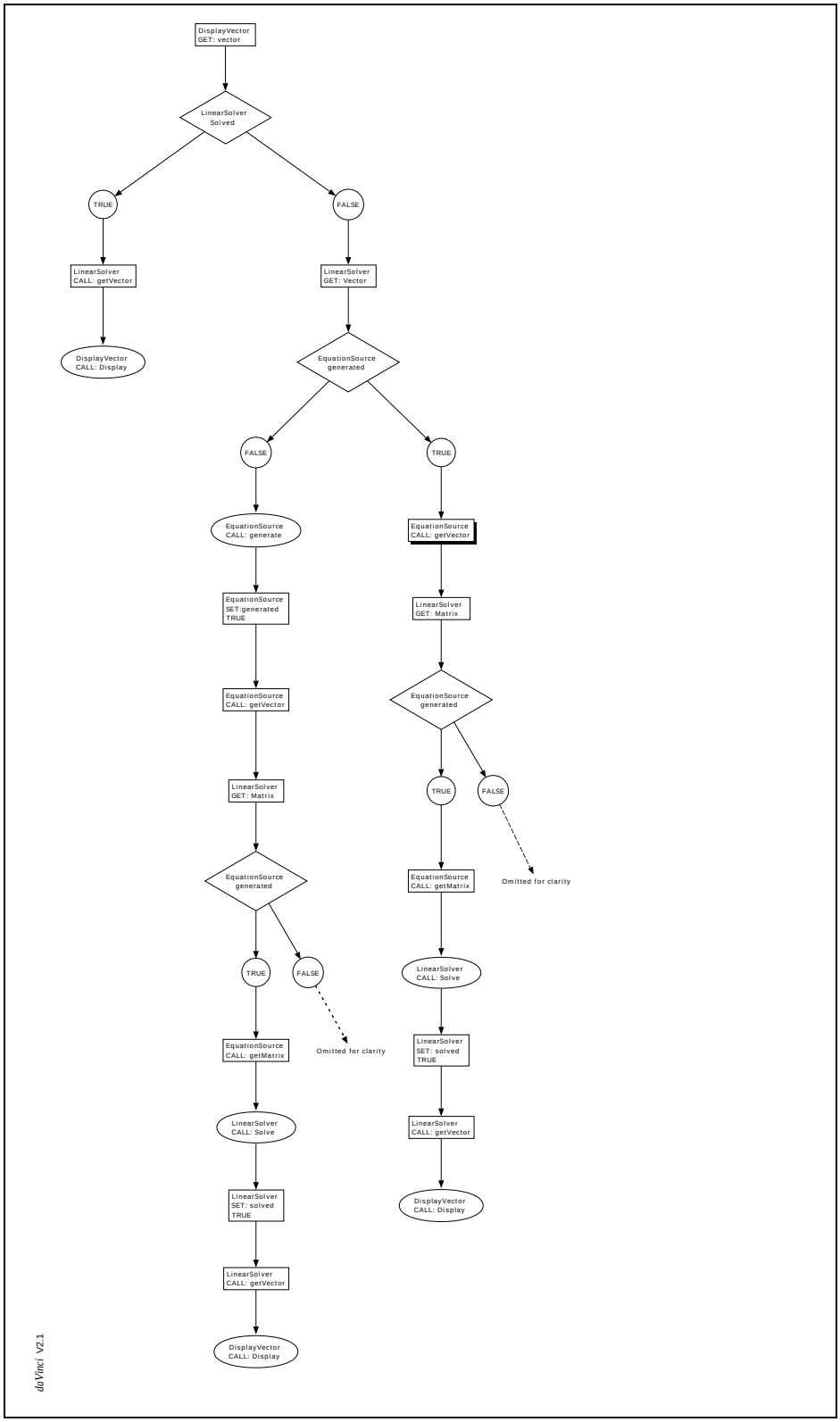

Fig. 6. Call Tree for Linear Solver

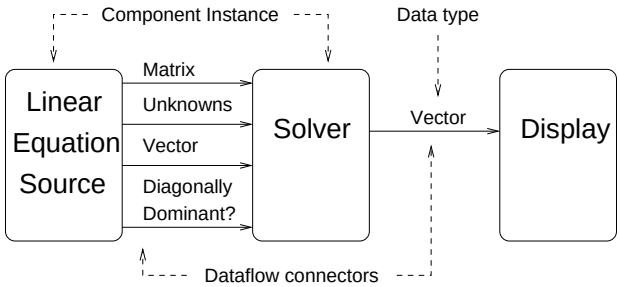

Fig. 7. Linear Equation Solver with Diagonal Dominance Information

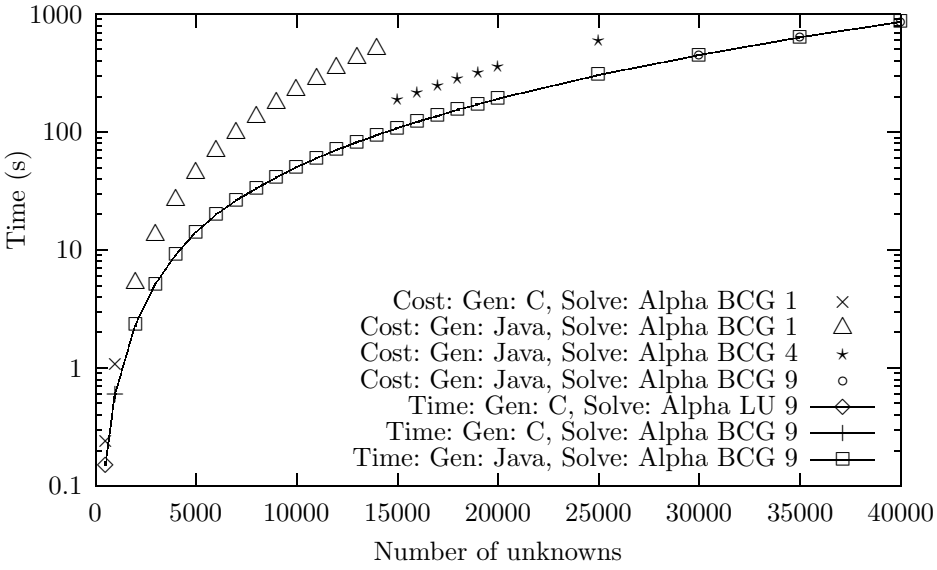

Fig. 8. Execution time for the composite application with different number of unknowns

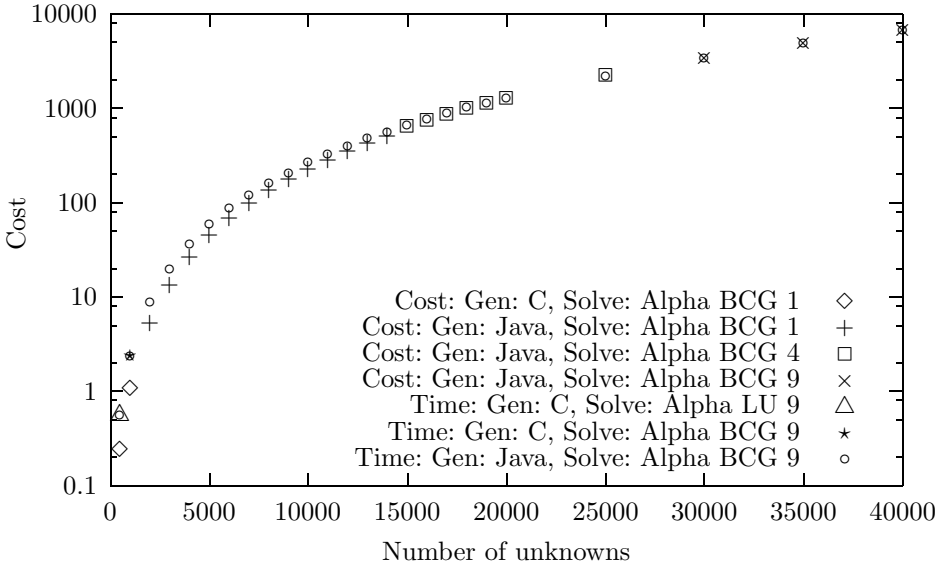

Fig. 9. Computational resource cost for the composite application with different number of unknowns