A Component Framework for HPC Applications*

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

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

Abstract. We describe a general component software framework designed for demanding grid environments that provides optimal performance for the assembled component application. This is achieved by separating the high level abstract description of the composition from the low level implementations. These implementations are chosen at run time by performance analysis of the composed application on the currently available resources. We show through the solution of a simple linear algebra problem that the framework introduces minimal overheads while always selecting the most effective implementation.

Keywords: Component Composition, Grid Computing, Performance Optimisation, High Level Abstraction.

1 Introduction

Within high performance and scientific computing there has been an increasing interest in component based design patterns. The high level of abstraction enables efficient end-user programming by the scientific community, while at the same time encapsulation encourages software development and reuse.

With the emergence of computational grids [1] it is increasingly likely that applications built from components will be deployed upon a wide range of heterogenous hardware resources. Component based design reveals both potential difficulties and opportunities within such a dynamic environment. Difficulties may arise where a component's performance is inhibited by the heterogenous nature of the environment. For example, an application consisting of tightly coupled components deployed across distant platforms will suffer performance penalties not present when executed locally. However the abstraction provided by the component methodology allows for a separation of concerns which enables specialisation and optimisation without reducing the flexibility of the high level design. Such resource and performance aware optimisation helps offset the difficulties of grid-based computation.

* Research supported by the EPSRC grant GR/N13371/01 on equipment provided by the HEFCE/JREI grants GR/L26100 and GR/M92455

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 540–548, 2001.

Algorithmic skeletons [2] have illustrated how a high level abstraction can be married to an efficient low-level high performance code, utilising the knowledge of the abstraction to optimise the implementation composition. We propose a component architecture that exploits this meta-data to optimise performance.

In this paper we describe an implementation of this architecture, which consists of an abstract component description language incorporating composition and performance information and a run time framework, together with a simple example of a component application and its native implementations. We first consider an overview of the architecture, and then examine different aspects of the framework.

2 Overview of the Component Architecture

We present a layered component architecture consisting of a high level language (an XML schema [3]), an execution framework and a component repository containing the component implementations and meta-data [4]. The run time framework executes the application, selecting implementations from the repository according to the meta-data, resource availability and the application description. The design and deployment cycle using this component framework is determined by its layered abstraction.

1. **Component Construction** Component interfaces are designed and placed within the repository, together with any implementations which may be written in native code. The developer places meta-data describing the implementation and its performance characteristics in the component repository.
2. **Problem Definition** The end-user builds an application within a graphical problem solving environment (PSE). The PSE is connected to the component repository and is used to produce a high level XML representation of the problem. This representation is implementation independent, using only the component interface information.

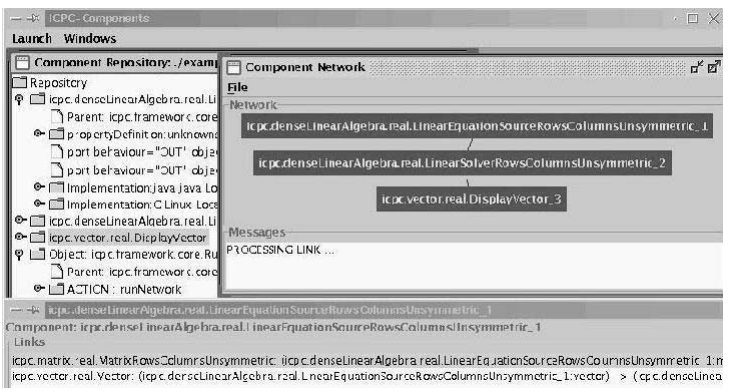

Fig. 1. GUI showing repository and composed application

3. **Run Time Representation** The XML representation is automatically converted into a set of executable Java objects, which act as the run time support for the application. This utilises the meta-data.
4. **Implementation Selection** The run time representation is deployed into an execution environment. Here available implementations are selected according to performance models and the available resources. High performance native code is loaded and executed by the run time representation. Implementation selection may occur more than once for a given component: changing circumstances, either through a changing resources or computation, may force the selection of a more efficient implementation.

3 XML Representation

The XML representation is the highest level of abstraction. The end-user composes the XML representation by means of an application builder tool. While we have provided a graphical user interface that enables rapid application development (see Figure 1), our Component XML (or CXML) provides a suitable target language for a number of possible end-user tools, without imposing constraints on their configuration.

```
<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>
```

The application builder produces a CXML application description document, as shown in the example above. This consists of the `<network>` and `<repository>` information. The network represents a composition of component instances together with any user customisations. The composition is a simple typed port and network mechanism, consisting of `<instance>` elements together with `<dataflow>` connectors. The connectors are attached between source and sink ports according to the component type. The user may customise the component by specifying simple values that are recorded as `<property>` elements.

The choice of components within the builder tool is determined by those contained within the repository. The repository CXML data provides the interface information for the `<component>` types, specifying `<port>` and `<property>` elements. Thus ensuring that connections in a network correspond to a meaningful composition. The types and default properties specified in the repository CXML allow customisation where required. The repository component information also indicates component inheritance and package information.

The repository also contains information needed to create the run time representation, namely the meta-data for the methods in the component implementation. These are represented by `<implementation>` and `<action>` elements. The

<action> elements specify the bindings to ports and the location of corresponding performance data. The package and location information for the executables is stored alongside the implementation data in the repository. These are referred to with <object> elements.

```
<repository>
 <component package="icpc.LinearSource" name="Source" version="1">
  <propertyDefinition type="external" name="degrees of freedom" value="1000"/>
  <port behaviour="OUT" objectPackage="icpc.Matrix" objectName="DgeRC" portName="matrix"/>
  <port behaviour="OUT" objectPackage="icpc.Matrix" objectName="Vector" portName="vector"/>
   <implementation language="java" platform="java" url="file:.">
    <action portName="matrix">
     <binding method="getMatrix">  ...  </binding>
     <classPerformanceModel type="initial" url="http:" />
    </action>
   </implementation>
   <implementation language="C" platform="Linux" url="file:."> ... </implementation>
 </component>
 <object package="icpc.Matrix" name="DgeRC" version="1">
  <method name="getMatrix" type="action">
   <argument mode="out" typeName="DgeRC" typePackage="icpc.Matrix" />
  </method>
 </object>
</repository>
```

4 Run Time Representation

When deployed onto a resource the CXML application description is converted into a *run time representation*, consisting of a network of Java proxy objects (JPOs), corresponding to the component instances in the application description document. This abstraction of the component application enables the dynamic selection of implementations, cross-component optimisation and implementation independent management.

Each JPO provides a black box abstraction of the component's implementation and acts as the run time interface for the component. Any interaction between components occurs at the level of the JPO. This provides a means to trap method calls and select the appropriate implementation when required.

The network of JPOs is created and linked by means of automatic code generation. The complete application description, including the relevant repository

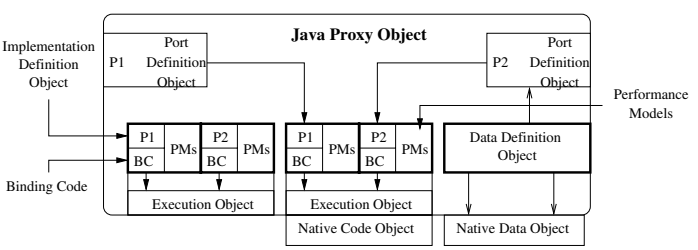

Fig. 2. A component's Java Proxy Object

data, is compiled from CXML into Java source code. The JPOs, together with the connections and property information, are represented by this 'glue code'.

The JPO contains Port Definition Objects, Implementation Definition Objects, Data Definition Objects and Execution Objects. See Figure 2.

Port Definition Objects (PDO) The CXML `<port>` element has an attribute 'type' that specifies whether it is an IN or OUT port. Each OUT `<port>` element has a corresponding PDO. Each IN `<port>` is represented by a reference to a PDO of the appropriate type. When the JPOs are created they are connected according to the application description by setting each IN port to refer to the appropriate PDO on the connected component. Each OUT `<port>` may offer many methods, which may then be called by any code within the component containing the corresponding IN port. It can be seen that this system provides a 'pull' model of computation.
During execution calls to a PDO are trapped, and an implementation is selected by examining the relevant IDOs.

Implementation Definition Objects (IDO) Each IDO is associated with a port. There are likely to be a number of different IDOs for each port, each representing a different implementation of the port's methods. The IDO contains the performance model meta-data for a given implementation, together with binding code that maps the ports methods to those of the actual implementation. When a method is called the PDO evaluates the relevant performance models and makes a decision as to which implementation to use. The binding code within the IDO maps the methods of the PDO to the actual implementation within the Execution Object.

Execution Objects (EO) Where an implementation is written in a native language (such as C), the execution object is a lightweight wrapper that makes use of the Java Native Interface [5]. This is responsible for the loading, unloading and execution of the native libraries. For pure Java implementations the EO acts as an intermediary between the actual implementation codes and the IDO. It should be noted that while there are possibly many IDOs for each port (corresponding to the different implementation options) there may also be many IDOs for a given EO. This is because a particular implementation may make use of shared objects and libraries appropriate for

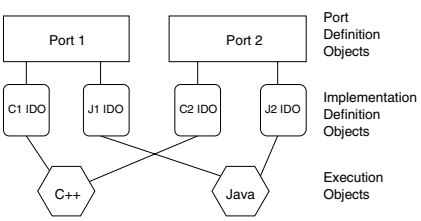

Fig. 3. Port, Implementation and Execution Objects

a number of different method calls from distinct ports. To maintain consistency a component will only select one EO at any given time. This is illustrated in Figure 3.

Data Definition Objects (DDO) To enable the migration of the component code between resources, and to allow the implementation selection to be altered during execution persistent data is stored outside the execution object. Thus the JPO needs to retain a reference to any data that may be used in subsequent method invocations. The format of the data clearly depends upon the implementation used to generate it. This reference together with mappings between the data format and Java are stored in a DDO.

Access to the persistent data is by method calls exposed in a PDO. By forcing calls to use the port mechanism, data access (often a significant part of a computation) is taken into account by the performance modelling and selection system. Where data is unavailable the PDO examines the respective IDOs as described above. Property values (which are stored as pure Java objects) are refered to by a DDO in the same way as any other data type. When the JPOs are created from the application description property values are assigned directly.

5 Implementation Selection

An application begins executing with an initial method call. The relevant PDO is responsible for the choice of implementation. By forwarding the relevant performance models from the IDOs to the framework a decision, based upon estimated running time, is made as to which implementation to use. The performance model usually refers to other calls the component needs to make. These calls are then forwarded to their respective PDOs, where they are trapped and the performance models returned. Thus a composite performance model for the application is built. The port mechanism exposes data movement between components allowing the data transfer time to be assessed and subsequently optimised to eliminate unecessary data copying. The technique is entirely deterministic, and results in a 'dry run' of the application. As each JPO may possess a number of possible implementations this procedure results in a call graph.

The performance model may also refer to data within the component, for example a property representing the number of entries in a matrix. As these

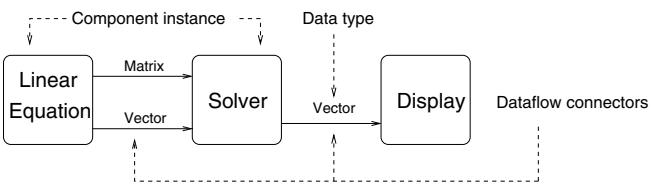

Fig. 4. Linear Equation Solver

properties may change during execution, or may not be known at start-up, the implementation decision can only take place during execution. The run time representation allows such a change of implementation with little overhead.

6 Example & Results

A composite application that solves a set of linear equations consists of three component instances; a simple linear equation generator, a solver, and a simple display component. See Figure 4. This example was used to provide the XML fragments already shown, and is now used to provide experimental results.

A set of real linear equations is generated from a component with a Java implementation. These equations are solved by one of two linear solvers written in C. The first is a direct solver using LU factorisation, whilst the second is an iterative Biconjugate Gradient method (BCG) [6]. These two algorithms have different time and memory requirements allowing the JPO to select the most appropriate implementation for the user-defined problem size. In this example the display component is only used to verify that the solutions are correct. All experiments were performed on a single 900MHz PC running Linux. The Java Native Interface [5] was used to allow access to the C implementations.

Figures 5 and 6 illustrate the performance models for the LU and BCG implementations. The performance models are generated by fitting curves to actual performance results. In both cases two curves are fitted to the data, each covering a range of problem sizes. Note that for the LU solver only the second curve is shown, as the first curve deals only with small matrix sizes and is not visible on the graph.

Figure 7 illustrates the execution time for the network, along with the solution times for a benchmark BCG and LU solver. For small matrix sizes the LU solution is more efficient, thus the JPO selects this implementation. It can be seen from the graph that both the network and LU execution times are close. For matrices of size 725 and larger the BCG method becomes more efficient and

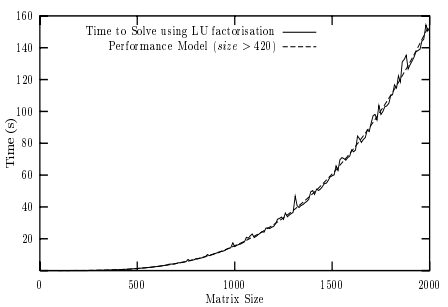

Fig. 5. Run time for LU solver

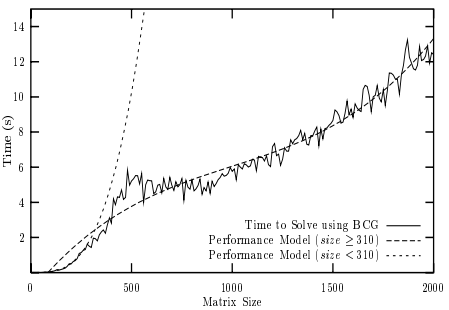

Fig. 6. Run time for BCG solver

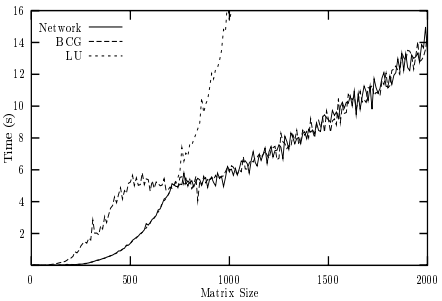

Fig. 7. Run time for the example system

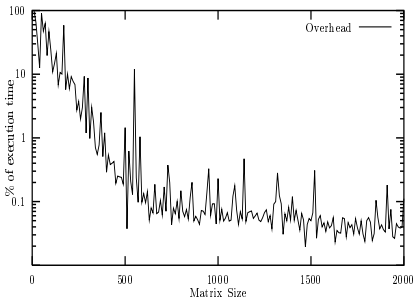

Fig. 8. Overhead from component network

the JPO selects this approach. This is again visible on the graph as the network results follow the BCG solution.

The overhead incurred from the use of the network solution is shown in Figure 8. This is the proportion of execution time that is not spent in computation. It can be seen from this graph that initially the overheads are high, with the value decreasing to less than 0.1% as the matrix size increases. The large overheads, and variations in results, for small matrices are partly due to the inaccuracy in timings of the system. Fluctuations in the BCG solution times can be attributed to the variable number of iterations required which effects the relative proportion of the overhead.

7 Conclusion

We have shown that the solution of a problem described as a set of interacting components can be implemented successfully. Furthermore it has been shown that the cost of performing the computation through such a framework is negligible, in comparison to the overall execution, when the problem size is large. In this case, the overhead appears to be independent of the problem size. Hence it is clear that the separation of concerns together with layered abstraction provides flexibility without sacrificing the performance HPC applications demand.

The techniques developed here are not restricted to the example system, but are applicable to arbitrary component networks and application domains. We are continuing to develop this framework for distributed parallel components.

References

1. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999. 540
2. J. Darlington, P. Au, M. Ghanem, and Y. Guo. Co-ordinating heterogenous parallel computation. In *Proceedings of International Euro-Par Conference*, pages 601–614. Springer, August 1996. Distinguished Paper. 541

3. W3 Consortium. XML: eXtensible Markup Language. `http://www.w3c.org/XML`. 541
4. S. Newhouse, A. Mayer, and J. Darlington. A Software Architecture for HPC Grid Applications. In *Proceedings of International Euro-Par Conference*, pages 686–689. Springer, August/September 2000. 541
5. R. Gordon. *Essential JNI, Java Native Interface*. Prentice Hall, 1998. 544, 546
6. William H. Press. *Numerical recipes in C : the art of scientific computing*. Cambridge University Press, 1988. 546