Building Ontology for Optimization and Composition of Parallel JavaBean Programs

Cheng-Wei Chen, Chung-Kai Chen, Jenq-Kuen Lee,
Department of Computer Science
National Tsing Hua University
Hsinchu 30043, Taiwan
{cwchen, ckchen}@pllab.cs.nthu.edu.tw and jklee@cs.nthu.edu.tw

Abstract— **In this paper, we propose an ontology specification for JavaBean programs, the object component model of Java. Our specification is written using the DAML+OIL language, which is based on the RDF schema and the XML syntax. The vocabulary of this ontology provides a basic terminology to annotate components with the information about the conditions and suggestions of adopting a component for component specializations. It also gives a reference critera for choosing the most suitable components at a given time and environment for performances and functionality purposes. With our design of the annotations, it's also possible to automatically retrieve the annotations of object components, connect them by their object-oriented relationships, organize them to form component databases, and discover them in the databases by component characteristics. This will facilitate the sharing of component resources in the internets. In addition, we also give an application scenario for employing this ontology specification. In our research work, we have been working on enabling the techniques for the run-time composition of parallel components. The employment of runtime compositions of components and the ontology specification proposed in this paper enable programs to adapt objects dynamically according to application and architecture characteristics. We demonstrate the employment of this ontology specification for adapting a parallel matrix component for performance purposes. This work provides a new direction for automatically enhancing long-running components by allowing components to improve and specialize themselves continuously in terms of performances and functionality.**

Keywords— **Ontology Specifications, Parallel Javabean Programs, Runtime Component Compositions, Code Specializations, Runtime Optimizations, Clustered EJB.**

I. Introduction

With the non-stopping spread of the network facilities, billions of computers have been connected to the internet to share their resources. The data distributed on this growing environments are so huge and diverse, it becomes an important issue to find an efficient way to use and adopt them. The technologies in mining the information and searching for useful data on the web are done mainly by searching engines to locate the data of interest. Recent progresses have also adopted the approach to connect the information in the web by their semantic relationship. The semantic web [12] is formed on this purpose. The semantic web approach is an attempt to give more standard terminology and ontology for application domains so that the catalog of products, the contents of literature, and the information for merchandise can be searched and analyzed in internet environments with precision.

The work was supported in part by NSC of Taiwan under grant no. NSC-89-2213-E-019-019, NSC-90-2213-E-019-016, NSC-89-2218-E-007-023, NSC-89-2219-E-007-012, and MOE research excellent project under grant no. 89-E-FA04-1-4.

In addition to the annotations and ontology descriptions for data in application domains, the building of ontology descriptions for software component are also important. We consider several possible scenarios below. First, application writers can describe the software specification for the software components in the design process of their software systems. If the ontology descriptions for components are annotated properly, many of the component module could be found via web search in the future. Due to object-oriented techniques, the component resources have common information associated with them, the interfaces they implements. Additional descriptions can be added. For example, a MP3 decoding component may have the version number information, the computing power needed for the lowest sound quality acceptable, the memory footprint of its implementation, and the license about its usage. These properties, we called metadata, address the difference between the implementations of the interface. It helps us search not only the kind of components, but also specific realizations. Next, let's consider another scenario. If the components of applications are allowed to be re-composed at runtime, the selection of a specialized component for specialized architectures, application characteristics, or additional functionality requirements can be done with proper annotations for component characteristics.

In this paper, we propose an ontology specification for JavaBean programs to attempt to address the issues above. Our specification is written using the DAML+OIL[5] language, which is based on the RDF[13] schema and the XML[11] syntax. The vocabulary of this ontology provides a basic terminology to annotate components with the information about the conditions and suggestions of adopting a component for component specializations. It also gives a reference critera for choosing the most suitable components at a given time and environment for performances and functionality purposes. With our design of the annotations, it's also possible to automatically retrieve the annotations of object components, connect them by their object-oriented relationships, organize them to form component databases, and discover them in the databases by component characteristics. This will facilitate the sharing of component resources in the internets. Our annotations are with ad-

ditional interfaces a component implements, and they are embedded into components so that the ontology interfaces annotated in our design can be investigated by Java reflection API's. In addition, we also give an application scenario for employing this ontology specification. In our research work, we have been working on enabling the techniques for the run-time composition of parallel components. The employment of runtime compositions of components and the ontology specification proposed in this paper enable programs to adapt objects dynamically according to application and architecture characteristics. We demonstrate the employment of this ontology specification for adapting a parallel matrix component for performance purposes. This work provides a new direction for automatically enhancing long-running components by allowing components to improve and specialize themselves continuously in terms of performances and functionality.

The remainder of this paper is organized as follows. Section II presents the component annotations. Next, Section III gives resource sharing scenario, and Section IV presents our proposed ontologies. and finally Section V presents experimental results.

II. The Presentation of Component Annotations

In this section, we give annotation mechanism for component writers to provide detailed characteristics for application components. The information can include the time complexity of the operation, the maximum latency of response, etc. They are known only by the programmers who had implemented them and should be annotated into components in the design and implementation stages. A convenient and flexible form to present such information is an important issue of component model design.

There are several ways to embed such data into a component. For example, to put them in the data fields of the component, to record them in a file associated with the component, or to have them be returned by the methods of components, etc. In our research, we propose to present the annotations as interfaces, called annotation interfaces, which are additional interfaces the components implement. For example, as shown in the RHS of Figure 1, the version number annotation can be represented as an interface called `Version` with a method `getVersion()`. The version number information is then returned by the result of invoking `getVersion()`. The first two diagams in Figure 1 give the annotations into a field or a method of a component, respectively. We do not adopt those two schemes in our proposed research framework. Our proposed annotation for components is mainly based on the following reasons:

1. **The annotations can be mutable.** The major reason not to store these annotation information somewhere is that the properties of the component may change dynamically. Using an interface or a method gives more dynamic extent for the specification of a component.

2. **The annotations can be grouped.** The interface design allows us to group the methods a component implements. We can define a class of interfaces that extends several annotation interfaces[9]. Thus they will contain all

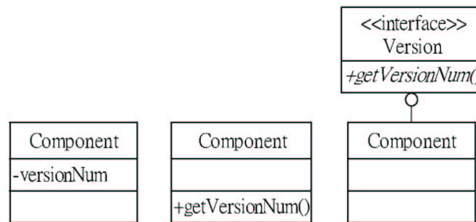

Fig. 1. Various ways to annotate a component.

the methods of the extended interfaces and work as an annotation collection. For example, the annotations usually found on the component generating animation frames can be grouped in a annotation collection called `Animation`, and the annotations usually found on the component handling audio codec can be grouped in a annotation collection called `Audio`. The annotation interface collections `Audio` and `Animation` can further form the `Video` annotation interface collection.

3. **The annotations can be analyzed.** The Java reflection API makes it possible to probe many design details of a component from the bytecode layer[6]. If we use the fields to store annotation information or methods to report annotation information, we need to apply some naming conventions to distinguish them with other component elements. In our design, we employ interfaces to represent annotations. The design enables all annotation interfaces to extend a common empty interface called `Annotation` for the denotation purpose, much like the `Serialization` interface does in the Java standard library. Therefore, Java reflection API can be used to analyze a component annotation interface automatically.

III. A Component Resource Sharing Scenario with Component Annotations

In the following, we give examples of component resource sharing scenario with component annotation interfaces.

A. The fetch of annotation information through the annotation interfaces.

Generally, an annotation information is a description of a characteristic. Through well-designed methods of the annotation interfaces, we give the component writer a way to reveal messages about his implementation. We can design a method that has an expressive name and a boolean return value to present the assertion of some facts. We use a method `isResumeSupported()` as an example. The component writer implements it to return `true` if his implementation commits to this fact. If the description gives values, it is more appropriate to put it in the method response. For example, if we want to say "this component need to be run on a 200MHz machine at least", we may design a method called `getRequiredCPUClockRate()` and the component writer implements it on his component to return 200. Numerical values, strings, or specific data structures can all be returned by a method.

Additional profiling methods can be added in designing the annotation interface. Consider the fol-

lowing scenario. The component relying on the network may have an annotation about the bandwidth preferred(lowest required). Then we can design an annotation interface `Bandwidth` having a method called `getPreferredBandwidth()`. Through invoking this method on the component we got the information of preferred bandwidth. Now we can give the bandwidth value of the current system and choose the right component. However, in fact the bandwidth the component actually got is a part of the total bandwidth of the system and it might vary depending on the sharings of the components on the network. Eventually, we will hope to profile and collect some information about the actual bandwidth for a component. This helps more precise adoption of the components. A possible solution to this problem is to provide an accompanied profiling method, `getAvailableBandwidth()`. It acts as a window for the component writer to deliver the profiling data. We then put this profiling method in the related annotation interface, which is `Bandwidth` in this example.

Once the annotation information are added into interfaces of components, they can be inspected. By invoking the methods of the annotation interfaces, we can get useful annotation information. Figure 2 illustrates the fetch of such annotation information. As Figure 2 shows, if the components are not currently running in a JavaBean programs, it can be deployed in a JavaBean container so that we can invoke methods on it. The JavaBean container is a program that can load JavaBeans and investigate or modify their properties by calling the corresponding methods. A JavaBean container serving as the component resource database/bank/pool can be practically built to carry out the queries on the annotation information of components.

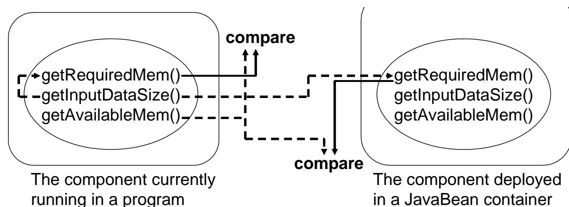

Fig. 2. An example of using methods in the annotation interface to fetch annotated information.

B. *The search of component specifications with annotations.*

In a component-based program, the programming logic is achieved by the interactions of components. These interactions are abstracted as interfaces in which the components have responsibilities to realize the functionality defined. In our work, we consider that all the functional interactions of a component are abstracted to one interface, called the source interface for the distinction from the proposed annotation interfaces. Theoretically, one component A can be replaced by another B if B implements all the methods in the source interface of A. Thus by comparing two components' source interfaces, this assertion can be claimed. However, given a source interface, we usually have many kinds of implementations each having specific characteristics. The adoption of wrong components may possibly cause dramatic performance drop, though the logic of the program is preserved. For example, when the input data size grows into thousands, the component containing a piece of code using bubble sort to sorting the data will become the performance bottlenecks in a program. This is why we need additional annotations to participate in the choice of components. In our design, we have the following search policy for components. We first locate the components who implement the required interface to guarantee the logical validity of adoption. We then fetch their annotation information about the appliance requirements, constraints, or suggestions. Finally, heuristic algorithms can be developed to pick up the most suitable component.

To find the components for a specific interface, we have to manage the source interfaces and components systematically in advance. Intuitively, they can be organized by their object-oriented relationship. As soon as we locate the target interface within the object-oriented hierarchy, all components and the components of their downstream are consequently the right ones to choose. These components are then the candidates for further searches. The annotation information of these candidates are examined and compared with the pre-given system configuration or runtime profiled information. The search and estimation of candidates can also proceed simultaneously in a BFS manner propagating from the queried interface to the bottom of the hierarchy.

.

IV. The Ontology

In the section above, we have shown the way to present the annotation information for components. However, a convention or a standard for giving the ontology information is needed. In this section, we give our proposed conventions and ontology classifications in annotating properties for components. Figure 3 gives a sample of our conventions for ontology represented in `DAML+OIL` language. We model the adoption knowledge of components in the following vocabulary. They are classified into five classes.

1. `SourceInterface` and `hasExtension` model the source interfaces and their extension relationship. They are used to build the source interface hierarchy and assert the logical validity of adoptions.

2. `hasImplementation`, `implement`, and `Component` are used to describe the implementation relationship between source interfaces and components. Through them and the source interface hierarchy, we can traverse candidates of specified interfaces.

3. `withAnnotation` and `AnnotationInterface` define the annotations of components.

4. `Denotation`, `Requirement`, `Preference`, and `Adminicle` model the annotation interface into four kind of annotations. `Denotation` is for the informational annotation. `Requirement` annotation gives the requirement on adoption. `Preference` provides adoption suggestion and

```
<rdf:RDF
  xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
>

<daml:Ontology rdf:about="">
  <daml:imports rdf:resource="http://www.daml.org/2001/03/
                                                    daml+oil"/>
</daml:Ontology>

<daml:ObjectProperty rdf:ID="Location">
  <rdfs:domain>
    <rdf:Alt>
      <rdf:li rdf:resource='SourceInterface'/>
      <rdf:li rdf:resource='Component'/>
      <rdf:li rdf:resource='AnnotationInterface'/>
    </rdf:Alt>
  </rdfs:domain>
</daml:ObjectProperty>

<daml:TransitiveProperty rdf:ID="hasExtension">
  <rdfs:domain rdf:resource='SourceInterface'/>
  <rdfs:range rdf:resource='SourceInterface'/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="hasImplementation">
  <rdfs:domain rdf:resource='SourceInterface'/>
  <rdfs:range rdf:resource='Component'/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="implement">
  <daml:inverseOf rdf:resource='hasImplementation'/>
</daml:ObjectProperty>

<daml:Class rdf:ID="SourceInterface">
  <rdfs:subClassOf>
    <daml:Restriction daml:minCardinality="1">
      <daml:onProperty rdf:resource='Location'/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>

<daml:Class rdf:ID="Component">
  <rdfs:subClassOf>
    <daml:Restriction daml:minCardinality="1">
      <daml:onProperty rdf:resource='Location'/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>

<daml:ObjectProperty rdf:ID="withAnnotation">
  <rdf:domain rdf:resource='Component'/>
  <rdfs:range>
    <rdf:Alt>
      <rdf:li rdf:resource='AnnotationInterface'/>
      <rdf:li rdf:resource='Adminicle'/>
    </rdf:Alt>
  </rdfs:range>
</daml:ObjectProperty>

<daml:Class rdf:ID="AnnotationInterface">
  <rdfs:subClassOf>
    <daml:Restriction daml:minCardinality="1">
      <daml:onProperty rdf:resource='Location'/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:ObjectProperty>
...
<daml:Class rdf:ID="Denotation">
  <rdfs:subClassOf rdf:resource='AnnotationInterface'/>
</daml:ObjectProperty>

<daml:Class rdf:ID="Requirement">
  <rdfs:subClassOf rdf:resource='AnnotationInterface'/>
</daml:ObjectProperty>

<daml:Class rdf:ID="Preference">
  <rdfs:subClassOf rdf:resource='AnnotationInterface'/>
</daml:ObjectProperty>

<daml:Class rdf:ID="Adminicle">
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="judgeCriterion">
  <rdfs:domain>
    <rdf:Alt>
      <rdf:li rdf:resource='Requirement'/>
      <rdf:li rdf:resource='Preference'/>
    </rdf:Alt>
  </rdfs:domain>
  <rdfs:range>
    <rdf:Alt>
      <rdf:li rdf:resource='Denotation'/>
      <rdf:li rdf:resource='Adminicle'/>
    </rdf:Alt>
  </rdfs:range>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="parameter">
  <rdfs:domain>
    <rdf:Alt>
      <rdf:li rdf:resource='Requirement'/>
      <rdf:li rdf:resource='Preference'/>
      <rdf:li rdf:resource='Adminicle'/>
    </rdf:Alt>
  </rdfs:domain>
  <rdf:range rdf:resource='Adminicle'/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="content">
  <rdf:domain rdf:resource='Denotation'/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="judge">
  <rdfs:domain>
    <rdf:Alt>
      <rdf:li rdf:resource='Requirement'/>
      <rdf:li rdf:resource='Preference'/>
    </rdf:Alt>
  </rdfs:domain>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="profiler">
  <rdf:domain rdf:resource='Adminicle'/>
</daml:ObjectProperty>

</rdf:RDF>
```

Fig. 3. The Ontology for Optimization and Composition of Parallel JavaBean Programs.

```
public interface RMI { ... }
public interface Latency extends Denotation {
    public String getLatency();
}
public interface ViplVersion extends Requirement {
    public String getRequiredViplVersion();
    public String getViplVersion();
}
public interface PacketSize extends Preference {
    public int getPreferredPacketSize();
    public int getAvgPacketSize();
}
public class VIARMI implements RMI, Latency,
                       ViplVersion, PacketSize {
    ...
}
```

Fig. 4. The `VIARMI` Component implements `RMI` and several annotation interfaces.

`Adminicle` represents the profilers that dynamically deliver useful information for the criteria of adoption judgement.
5. `judgeCriterion` and `parameter` define how the information from `Adminicle` be used as the criteria of the adoption and the parameter of other annotation.

The vocabulary is formally defined in Figure 3 using `DAML+OIL` language.

In the vocabulary above, class 1, 2, and 3 are terms directly mapped from the object-oriented design and denoted for the relationship related to the source interfaces and the components. This information can be automatically generated Java reflection API. For class 4 and 5 above, a naming convention of the annotation interfaces can help translate the methods into the ontology model. Thus given a component with annotation interfaces implemented, We can generate an annotation description in `daml` format.

V. Application Examples

A. RMI Component

Java RMI [7] is an important fucntion supported in Java standard library as the communication basis for distributed computing. In addition to the inherent implementation of RMI using `TCP/IP` sockets, we can also implement RMI over various network architecture. Note that one major part of RMI functionality is carried out by `UnicastRemoteObject`. The RMI remote objects export themself to provide RMI service by calling `UnicastRemoteObject.exportObject()`. Suppose we have reimplemented `UnicastRemoteObject` to have RMI been connected over VIA, an user-level network architecture. We can have a possible annotation interface shown in Figure 4. Figure 5 then shows the annotation description from `VIARMI` in `daml` format. Note the `daml` format can be automatically generated from annotation interface. Various RMI implementations such as RMI over bluetooth, RMI over nexus, RMI over IB can be annotated with specifications so that the components can be used or adopted for specializations.

B. Experiments with Parallel Matrix Component

```
<rdf:RDF
  xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:cao="http://.../cao#"
>
<cao:Component rdf:ID="VIARMI">
  <daml:versionInfo>$1.0$</daml:versionInfo>
  <cao:Location>...</cao:Location>
  <cao:implement rdf:resource='RMI'/>
  <cao:withAnnotation>
    <rdf:Bag>
      <rdf:_1 resource='Latency'/>
      <rdf:_2 resource='RequiredViplVersion'/>
      <rdf:_3 resource='GetViplVersion'/>
      <rdf:_4 resource='PreferredPacketSize'/>
      <rdf:_5 resource='GetAvgPacketSize'/>
    </rdf:Bag>
  </cao:withAnnotation>
</cao:Component>

<cao:Denotation rdf:ID="Latency">
  <cao:Location>...</cao:Location>
  <cao:content>220</cao:content>
  <rdfs:comment>microsecond</rdfs:comment>
</cao:Denotation>

<cao:Requirement rdf:ID="RequiredViplVersion">
  <cao:Location>...</cao:Location>
  <cao:judge>getRequiredViplVersion()</cao:judge>
  <cao:judgeCriterion rdf:resource='GetViplVertion'/>
</cao:Requirement>

<cao:Adminicle rdf:ID="GetViplVersion">
  <cao:profiler>getViplVersion()</cao:profiler>
</cao:Profiler>
<cao:Preference rdf:ID="PreferredPacketSize">
  <cao:Location>...</cao:Location>
  <cao:judge>getPreferredPacketSize()</cao:judge>
  <cao:judgeCriterion rdf:resource='GetAvgPacketSize'/>
</cao:Profiler>

<cao:Adminicle rdf:ID="GetAvgPacketSize">
  <cao:profiler>getAvgPacketSize()</cao:profiler>
</cao:Profiler>
<cao:SourceInterface rdf:about="RMI">
  <cao:Location>...</cao:Location>
  <cao:hasImplementation rdf:resource='VIARMI'/>
</cao:SourceInterface>

</rdf:RDF>
```

Fig. 5. A generated annotation description file of Component `VIARMI`.

The ontology specification can be used to adapt components at runtime. In our research work, we have been working on enabling the techniques for the run-time composition of parallel components. The ontology specification proposed in this work helps adapt objects dynamically according to application and architecture characteristics. Our runtime support for runtime component compositions is based on the dynamic proxy support of Java[8]. As Java is a statically typed language, objects are created in heap and held by reference variables with a particular type. If we want to re-compose an object, a new object with the same type of subtype can be created in heap and assigned to the original reference variable. However, an object can be referenced by more than one reference variable. In addition, the references can be forwarded as arguments in method invocations. Therfore, assigning an object to a reference can

```
public interface IMatrix { ... }
public interface JPVMVersion extends Requirement {
    public int getRequiredJPVMVersion();
    public int getJPVMVersion();
}
public interface Sparsity extends Preference {
    public int getPreferredSparsity();
    public int getSparsity();
}
public interface ParaNodeNum extends Preference {
    public int getPreferredParaNodeNum();
    public int getAvlParaNodeNum();
}
public class DMatrix implements IMatrix, Sparsity {
    ...
}
public class SMatrix implements IMatrix, Sparsity {
    ...
}
public class PDMatrix implements IMatrix, Sparsity,
                         JPVMVersion, ParaNodeNum {
    ...
}
```

Fig. 6. The Parallel Matrix Components implement `IMatrix` and a variety of annotation interfaces.

not alter other references pointing to the original object. To avoid problems in component substitution, the proxy object in Java is needed to wrap the original object. Therefore, the reference variables maintain their references to the proxy object, and assigning an object to the proxy object can alter other references to the original object through the proxy object.

In our experiment, we demonstrate the benefit of employment of this ontology specification and runtime compositions for a parallel matrix components on a 8-node PC-cluster. The 8-node PC-cluster is connected with 100Mbps ethernet, and each node is running on 800MHz AMD Athlon CPU with 256MByte memory. The software environment includes SUN J2SDK 1.4.0-rc, PVM 3.4.4, and jPVM 1.1.4 over Linux kernel version 2.4.8. The software is wrapped with Java components and JavaPVM interface for experiments.

The parallel matrix sets include dense matrix, sparse matrix, and parallel matrix. We use a conjugate gradient solver for experimenting our matrix components. The dense matrix, sparse matrix, and parallel matrix are all equipped with ontology annotation interface, as shown in Figure 6. In the runtime execution, if a dense matrix is executed for a long time, one might adapt the dense matrix component into parallel matrix component by utilizing the ontology annotations. Similarly, in the runtime execution, if a dense matrix is executed for a while and one finds the dataset for the matrix is sparse, one might adapt the dense matrix component into a sparse matrix component by utilizing the ontology annotations. The decisions of the adapting a component can be done by a runtime monitor facility or container, while the re-composition is implemented by Java proxy in our system. Table I first gives the benefits of switching a dense matrix component into a sparse component on CG solver. The data set is actually a sparse dataset. If a adaption can be done after initial iterations, the performance can be gained for the rest of iterations. The dataset is a sample from the toolbox of Matlab and a well-known test case[2, page 352, Fig. 4][1]. Potential benefit exists in such a sample example. In addition, we can also observe the overhead in our proxy implementation. Table II shows another test example. Again after initial iteration on sequential dense matrix component, an adaption can be made to switch the component into a parallel dense matrix component. Table II gives the potential benefits for such cases.

TABLE I
THE COMPUTATION TIME (MS) OF CONJUGATE GRADIENT SOLVER ON DENSE MATRIX AND SPARSE MATRIX, THE TEST CASE IS A 479*479 SPARSE MATRIX.

	Direct Ref.	Dynamic Proxy
DenseMatrix	1004.9	1010.1
SparseMatrix	111.4	111.7

TABLE II
THE COMPUTATION TIME (MS) OF CONJUGATE GRADIENT SOLVER ON PARALLEL MATRIX, THE TSETCASE IS AN 800*800 DENSE MATRIX.

Procs.	Direct Ref.	Dynamic Proxy
1	551.2	559.1
2	287.5	292.2
4	155.9	172.0
8	100.0	100.8

REFERENCES

[1] Rong–Guey Chang, Cheng–Wei Chen, Tyng–Ruey Chuang, and Jenq Kuen Lee. Towards automatic supports of parallel sparse computation in Java with continuous compilation. *Concurrency: Practice and Experience*, 9(11):1101–1111, November 1997.

[2] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333—356, January 1992.

[3] T. R. Gruber. *Toward principles for the design of ontologies used for knowledge sharing*, In Nicola Guarino, Ed., International Workshop on Formal Ontology, Padova, Italy, 1993.

[4] U. Holzle., *Integrating independently-developed components in object-oriented languages.*, In ECOOP, pages 36–56, 1993.

[5] Reference description of the daml+oil (march2001), *Ontology markup language*, http://www.daml.org/2001/03/reference.html, March 2001.

[6] Sun Microsystems, Inc. *Java Core Reflection API and Specification*, Available at http://java.sun.com/j2se/1.4/docs/api/java/lang/reflect/package-summary.html.

[7] Sun Microsystems, Inc. *Java Remote Method Invocation Specification* , revision 1.70 edition, December 1999. http://java.sun.com/products/jdk/rmi/index.html.

[8] Sun Microsystems, Inc. *Dynamic Proxy Class API*, Java 2 SDK Std. Ed. Documentation v1.3.1 Available at http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html

[9] Sun Microsystems, Inc. *The Java Language Specification*, Second Edition. Available at http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html.

[10] Gruber, T.R. (1993). *A Translation Approach to Portable Ontology Specifications*, Knowledge Acquisition, Vol.5, pp.199-220.

[11] W3C. *Extensible Markup Language (XML)*, 2001. Available at http://www.w3.org/XML/.

[12] W3C. *Semantic Web*, Available at http://www.w3.org/2001/sw/, February 2001.

[13] W3C. *Semantic Web Activity: Resource Description Framework (RDF)*, 2001. URL: http://www.w3.org/RDF/.