Scalable Computing: Practice and Experience
Volume 6, Number 1, pp. 17–32. http://www.scpe.org
ISSN 1895-1767

AD HOC METACOMPUTING WITH COMPEER

KEITH POWER* AND JOHN P. MORRISON*

Abstract.
Metacomputing allows the exploitation of geographically seperate, heterogenous networks and resources. Most metacomputers are feature rich and carry a long, complicated installation, requiring knowledge of accounting procedures, access control lists and user management, all of which differ from system to system. Metacomputers can have high administrative overhead, and a steep learning curve which restricts their utility to organisations which can afford these costs.

This paper describes the Compeer system, which attempts to make metacomputing more accessible by employing an implicitly parallel computing model, support for programming this model with a Java-like language and the construction of a dynamic *ad hoc* metacomputer that can be temporarily instantiated for the purpose of executing applications.

Key words. Peer-to-peer, decentralised, metacomputer, ad hoc metacomputing, graph, java, translation, condensed graphs

1. Introduction. The variety of metacomputers has grown considerably in recent years. There are batch processing systems, like CODINE [21] and PBS [7]. There are metacomputers designed with specific types of application in mind, like DiscWorld [20] and AppLes [2]. There are toolkits to facilitate building tailored distributed applications, like Globus [5], and there are systems which present a network as a single virtual machine (VM), like Legion [13]. There is even a system, Everyware [22], which links these metacomputers together.

These systems have tremendous flexiblity of operation, allowing different sites to link and share resources, to store detailed accounting information to ensure resources are shared equally, or to charge an organisation for its usage. These features are a consequence of the large user base and the permanence of the associated systems.

Exposure to metacomputing systems comes also in the form of popular metacomputing applications such as SETI@Home[19] and distributed.net [4]. Configuration and setup costs are kept to a minimum using a simple download and installation procedure. From the participant's perspective, CPU cycles are contributed to solving a single, long running application. Since compute tasks cannot be submitted by a participant to the metacomputer, the benefit of these systems is unidirectional, relying on the altruism of the volunteers.

Compeer does not have any preexisting, or necessarily permanent, architectural component or communications topology. Rather, it is dynamically constructed in an ad hoc manner by the coming together of multiple "peer" machines. Peers join the collective through either a process of discovery or introduction.

The advantage of the Compeer system comes from the fact that it is easily and inexpensively deployed similarly to the popular metacomputing applications, yet it is a metacomputing platform giving each peer the facility to contribute to and benefit from the association. The remaining sections of this paper will discuss the Compeer program methodology, an overview of the Compeer architecture and possible deployments and finally present two sample applications.

Compeer executes applications in the form of Condensed Graphs ($\mathcal{CG}$s). The $\mathcal{CG}$ model of computing is a very expressive, implicitly parallel model which allows different sequencing constraints to be specified in applications. It provides automatic synchronisation, and is hierarchical, facilitating the easy distribution of work across a network. The sequencing facilities of $\mathcal{CG}$s are not relevant here, so for the purposes of this paper $\mathcal{CG}$s can be considered as similar to Dataflow [9] graphs.

Applications can be translated from Java[6] to an equivalent $\mathcal{CG}$ form to provide automatic parallelism. Java was chosen because it has a large user base and a well defined, publicly available grammar. An Object Oriented language was chosen to facilitate the easy integration of objects into $\mathcal{CG}$ applications

2. Overview of Compeer.

2.1. Architecture.

Compeer is a Java based peer-to-peer metacomputing platform. Each peer in the metacomputer is a Compeer object. These objects, linked together, form the metacomputer. It is possible to have more than one Compeer object running per machine, but for the purpose of this paper, it is assumed that each machine runs a single Compeer object, and so the term peer is applied interchangeably to both a Compeer object and the machine on which it resides. Being peer-to-peer, each peer is equal in ability and communicates symmetrically with other peers. Since the system is serverless, there is no single point of administration,

*Computer Science Dept., University College Cork, Ireland (k.power@cs.ucc.ie, j.morrison@cs.ucc.ie)

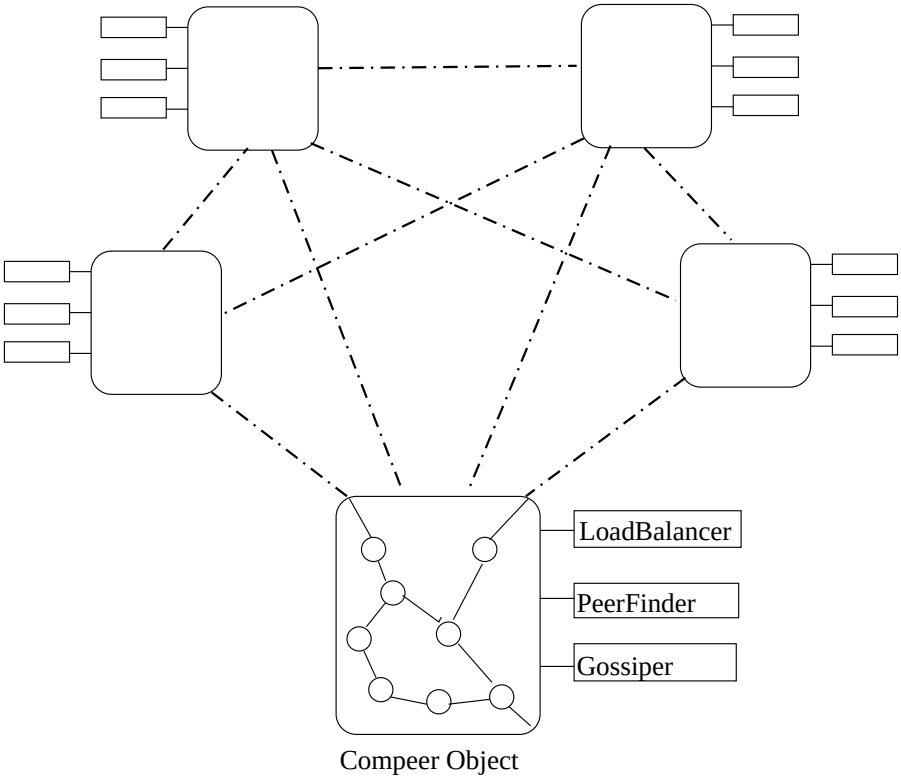

FIG. 2.1. The Compeer Architecture. A system of 5 peers is illustrated with $\mathcal{CG}$s running on each peer and communicating with each other. The Compeer object on each peer is responsible for gathering system information, for creating objects corresponding to incoming $\mathcal{CG}$s and for initiating the creation of $\mathcal{CG}$s on other peers.

which enables the construction of cooperative metacomputers by willing participants facilitating the sharing of resources.

Each Compeer object is supplied with several helper objects at startup: a PeerFinder object, a LoadBalancer object and a Gossiper object. The PeerFinder object is responsible for discovering other peers and keeping references to them. The LoadBalancer object is responsible for deciding where in the network new nodes should be created, based on a list of peers and their relevant statistics e.g. CPU load. The Gossiper object is responsible for inter-peer communications, for example, spreading results around the network when a computation is complete, which also requires a list of peers. Three java interfaces are used, rather than classes, facilitating the easy replacement of any of the components.

This modular approach yields a versatile metacomputing system that can be used in a variety of deployments, from intra- to inter- net. For example, in an intranet deployment where the number of machines is small, a PeerFinder which discovers other peers via multicast is viable and can be used. Over the Internet, a PeerFinder which reads the IPs of other peers from a webpage can be implemented. In small deployments, each peer can hold a reference to all other peers in the network, and so a very simple Gossiper can be used to propogate messages. With a large network, each peer may contain references to only a portion of the other peers in order to allow scaling. This would require a more complex Gossiper.

Compeer metacomputer can also be constructed in an ad hoc fashion, with peers connecting and leaving the network at will. This method makes several useful deployments possible, for example a situation where several researchers run their own peers and direct them to the other peers, allowing them to share their resources.

2.2. Program Execution. Compeer executes programs in the form of $\mathcal{CG}$s, described in a graph description language. $\mathcal{CG}$s are hierarchical, so a single application consists of descriptions of multiple graphs. As a graph executes, each node in the graph is represented by a CGNode Java object in Compeer. Specific nodes such as the Filter node are subclasses of an abstract class, CGNode. This nodes implements the $\mathcal{CG}$ firing rule, which means that a node which has its required inputs in the correct form and has a destination can fire. In

this implementation, resources permitting, any node that can fire, fires immediately. $\mathcal{CG}$ nodes in the graph are represented in Compeer by a $\mathcal{CG}$ object, which contains a variable detailing the KDLang instructions it will use when firing. When a $\mathcal{CG}$ node fires, it's instructions are interpreted, leading to the creation and linking together of nodes representing that $\mathcal{CG}$. This permits a simple approach to program execution: create a single CG object, supply it with the graph instructions and give it an X node as a destination. The CG node will fire to create the objects corresponding to the nodes in a graph and link them together as represented by the graph. Any fireable nodes will fire, placing their results on the input ports of other nodes, causing them to fire and so on, until a result is produced and fired through the Exit node ending the computation

Nodes can be created on the local peer or on any other known peer. The decision of where to create them is the responsibility of the LoadBalancer object. A peer can run many applications simultaneously, and the nodes in each application are separate and comprise a *process*. Before a peer can send nodes for a process to other peers it must 'introduce' those peers to the process by sending them the graph description of the application being executed.

2.3. Primitive operations in Compeer. Earlier implementations of the $\mathcal{CG}$ platform used functions as operators, or primitive nodes, to accomplish data transformation e.g. the plus node in (Fig. 4.1) corresponds to a plus function at execution time.

In Compeer data transformation is accomplished via method invocations on objects. Correspondingly, Compeer requires nodes which can represent the creation of these objects, and invocations upon them. Java, DCOM and CORBA objects are currently supported via Compeer plugins, though plugins to support other object technologies can be developed. Basic type conversion between different object technologies is handled by Compeer.

To support objects as node primitives, as opposed to functions like `+` in the factorial example, Compeer introduces two nodes, `create` and `invoke`.

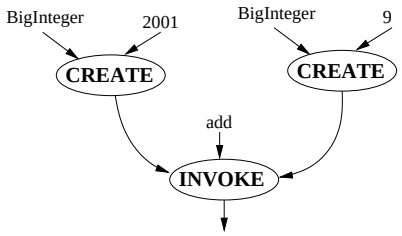

FIG. 2.2. Example of `create` and `invoke` nodes

The `create` node is used to dynamically create any Java object. It takes the name of the class to create as a parameter, and optionally, parameters for a constructor. There are also `createDCOM` and `createCORBA` nodes, although it is also possible now to create CORBA objects using the libraries supplied with the Java Development Kit 1.4.

The `invoke` node is used to dynamically invoke methods on an object. It takes an object reference, a method name and optionally parameters. In cases of method overloading the method to execute is determined using the parameter types. Similarly, there are `invokeDCOM` and `invokeCORBA` nodes.

In (Fig. 2.2), both `create` nodes will create instances of the `BigInteger` class and initialise them with 2001 and 9. The `add` method on one of these instances will be invoked using the other as a parameter.

The $\mathcal{CG}$s that Compeer executes consist of nodes representing the creation and execution of Java, CORBA and DCOM objects. In fact, Compeer can be viewed simply as a distributed scheduler of these objects.

2.4. Reflection and Polymorphic nodes. A Compeer object can itself be manipulated by a graph it is executing. This possibility arises because Compeer is implemented as a Java object. A graph which obtains a reference to the Compeer on which it is running can invoke methods on it to obtain information or manipulate other graphs. This is useful for cases where an application needs to know how many peers are connected to the peer it's running on, or how busy a peer is.

Nodes in Compeer are also objects, therefore they can be manipulated in graphs too. This can be used to set a $\mathcal{CG}$ nodes instructions at run time. A node whose instructions are altered during the course of its existence is termed *polymorphic*.

Programmatic dynamic load balancing can be achieved by combining these two techniques. For example, if an application consisting of 100 resource intensive $\mathcal{CG}$s were executed on a single peer and each of the individual $\mathcal{CG}$s were handed out to peers, then the first peer must distribute 100 nodes itself. This is analogous to the master/slave concept of work distribution. If, at run time, the graph tailored it's behaviour to the number of peers available, better results could be achieved. For example, after interrogating the Compeer object and discovering there are 5 peers connected whose loads are low enough to be useful, instructions could be produced for a node that when fired would distribute a $\mathcal{CG}$ containing 20 of the tasks. This process could be repeated on those peers to facilitate the rapid distribution of work and take advantage of the topology of the network.

This technique allows the construction of $\mathcal{CG}$ applications which can tailor their behaviour to the underlying architecture at run time.

It is also possible for Compeer to alter the description of the graph it is executing to optimise performance. For example, an idle Compeer could alter certain connections in the graph to make more instructions eager, and thus take more advantage of the CPU. Another example would be where Compeer measures the execution of a method invocation and alters the graph to reflect the measurement. In this way feedback from the executing graph can alter it's future execution. Only the order and timing of execution is changed, the program remains the same.

3. Examples of Deployment. Compeer can be deployed in a variety of situations, due to it's minimal configuration, decentralized architecture and object based interface. Three such deployments are presented now.

Compeer is completely decentralized and therefore has no single point of administration. While a single point of administration is generally considered advantageous in networked systems, the lack of one here allows the metacomputer to be deployed in an ad hoc fashion, that is, the metacomputer can be formed by peers, located anywhere, joining and leaving at any time. This allows a cooperative metacomputer to be built among willing users, where each user runs their own peer on their own machine. To connect into the network they need know only the IP of one other peer, from which their Compeer can obtain references to other peers. Since Compeer is Java based, each user can customise their security policies to enable as much sharing of resources as they wish, so there is no need for a global administrator of the system. This type of arrangement enables researchers to construct applications capable of easily sharing data and resources with their peers.

To deploy Compeer it is sufficient to supply each peer with a list of IP/name pairs it can use to find other peers. These can be supplied as a file when each Compeer is executed, or placed on a web page where they can be downloaded. This simple installation enables another type of ad hoc deployment, where a single user can start a Compeer on multiple machines to form a usable metacomputer. This is useful in cases where a user has an application which takes a lot of CPU time which they want to finish sooner. Rather than setting up and configuring a metacomputer, which may take longer than the application, they can deploy Compeer quickly and execute their application immediately. If necessary, Compeer can automatically uninstall when the application ends. Compeer can be started and stopped quickly and easily, reducing the cost of investment associated with setting up a metacomputer.

Each peer is an object, which makes Compeer versatile in terms of interfacing. An example deployment which exploits this would place peers on a network in any arrangement, including the two previously described, but would be accessed via a webserver. An internet user could visit a webpage which is generated dynamically, for example, a Java Server Page or Servlet. This page could interface with any Compeer in the network, submit some work, retrieve the result, parse it and send it to the users web browser to display. This enables users to easily construct web based interfaces to their network, or to distribute the load of their webserver across their network.

The system is written in Java, so it can operate on Windows, *nix and any other operating system for which a JVM is available. It can be configured to run in the background on *nix systems, or as a service on Windows, to permanently install it.

3.1. Note on security. The system leverages operating system security and a simple password system to prevent unauthorised users submitting work to the metacomputer, or receiving work or results from it.

Where one user has configured the system, Compeer objects running on each machine execute in that users process space. Where several people set up the system, there is a Compeer object running on each of their machines, in their process space. It is assumed that these people trust each other. In cases where trust is not

absolute, individual users can tailor their *java.policy* file to restrict other users access to their machine e.g., to prevent file system access and/or to deny loading of DLLs. The operating systems separation of process space prevents unauthorised users from tampering with the running Compeer objects.

It is possible for malicious users of the system to produce fake results. Currently, no effort is made to detect this or protect against it. Several methods of protection in a master/slave arrangment have been put forward [17]. The problem is compounded in a peer-to-peer metacomputer. One approach to sabotage tolerance is to distribute multiple copies of a piece of work to different slaves. If the slaves return different results, then one of them must be falsifying it's data, and it's identity can be ascertained by further investigation. Applying this approach in Compeer situation is infeasible: a peer passing a $\mathcal{CG}$ to one peer could instead pass duplicates to several peers. Any $\mathcal{CG}$s in that application will also be duplicated by each peer, and so on, leading to an exponential increase in the amount of work to be done at each stage, rendering the distributed execution of any non-trivial application counterproductive.

Digital signatures have been used to verify the identity of participants [18]. A method of producing an audit trail for computations using digital signing is being investigated. It is envisaged that this will allow the identities of peers who produce incorrect results, enabling a punishment policy, such as blacklisting, to be introduced.

3.2. Usage. Compeer is completely decentralised, and so jobs can be submitted to any peer in the network for execution. The metacomputer has a versatile interface enabled by the fact that each peer is an object, and each peer is equal. Once a reference to any peer is obtained, currently via RMI, the peer can be manipulated, examined or sent work. It is also possible to obtain references to its known peers, and so on, until a complete picture of the network is built up. This ease of manipulation leads to a wide variety of usage scenarios. Work can be submitted to a peer via a graphical or command line shell. The metacomputer can be invoked from a JSP to provide dynamic content for webpages, or invoked using XML/SOAP to enable web services. GUIs can be built whose buttons result in the execution of complex distributed applications across a network via a single, simple method call.

For non-programmatic access, a user interfaces with the Compeer metacomputer via a graphical shell. A shell can connect to any Compeer object, though in practice it is more efficient to connect to the Compeer object running locally. A text file representing a $\mathcal{CG}$ application is loaded using the shell and is submitted to the system. If debugging is disabled, the application executes quietly until a result is returned.

If debugging is enabled, a list of peers participating in this process is displayed in the shell. For each peer, a list of any of the current process's nodes present on that machine is listed. These nodes can be selected to highlight their inputs and destinations. The computation can proceed forward in a step-by-step fashion. A global step informs all Compeer objects to execute the next instruction available in this process. An individual step directs a single Compeer object to execute its next available instruction for this process. Alternatively, individual nodes can be fired and their properties inspected. Thus, per-process, per-peer and per-node debugging is possible. Debugging a process has no effect on any other executing Compeer processes.

Debugging is useful to users with knowledge of $\mathcal{CG}$s. Knowledge of $\mathcal{CG}$s and how they execute in Compeer is not a prerequisite for using this system, since Java programs can be converted into suitable graphs, though it is recommended for users interested in optimising their applications performance.

3.3. Note on Application Development. $\mathcal{CG}$ applications are written in KDLang, a simple language used for constructing graphs. They can be converted to XML for use with other graph based applications, in particular [14]. Development of these applications using KDlang can be time-consuming and error-prone. To ease this process, programs can be developed graphically using a drag-and-drop development environment, or via a visual KDLang editor (Fig. 3.1), which displays the graph as it's being described. Large applications can be produced by converting Java to KDLang, and then optionally tailoring the generated code in the visual editor.

4. KDLang grammar. The KDLANG language consists of instructions for building condensed graphs. There are instructions to create nodes, to connect them in various ways (either a stem or graft to facilitate eager or lazy execution), and instructions to set the value of simple nodes. An example of the instructions needed to create the graph in (Fig. 4.1) follows:

```
CREATE PLUS addNode
CREATE SIMPLE a
CREATE SIMPLE b
```

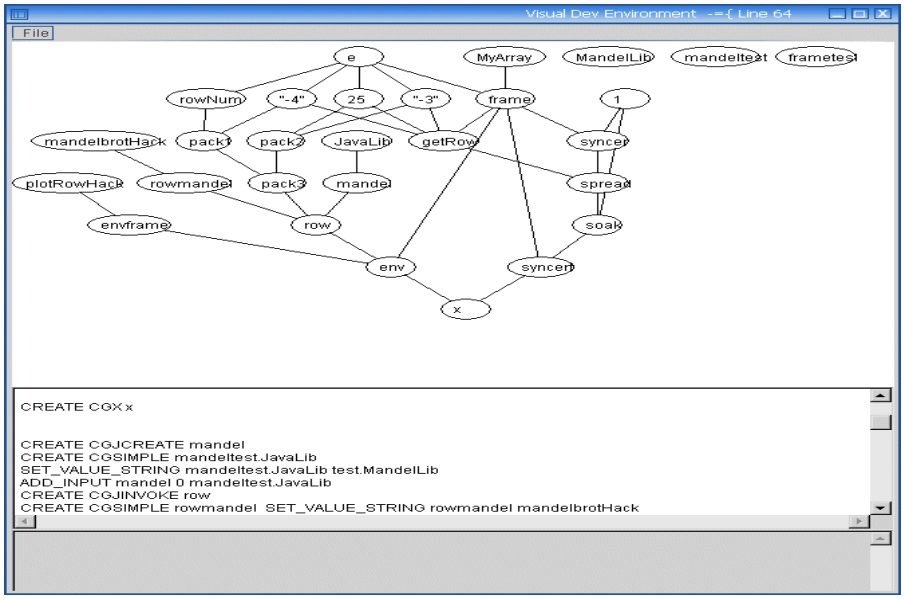

FIG. 3.1. Visual Development Environment. As the KDLang graph description is entered into the center panel, the graph is constructed and displayed in the top panel. Any errors in the KDLang are reported in the bottom panel.

```
CREATE X result
SET_VALUE_INTEGER a 5
SET_VALUE_INTEGER b 7
ADD_INPUT addNode 0 a
ADD_INPUT addNode 1 b
ADD_DESTINATION addNode result 0
```

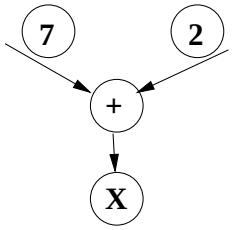

FIG. 4.1. *Simple CG to add two integers. Written without Create and Invoke nodes.*

4.1. Definition of KDLang. The KDLang grammar is defined using the ANTLR[15] syntax rules.

```
class P extends Parser;

programDef
   :  (cgDef)+ EOF
   ;
cgDef
   :  WORD NUMBER (statement)* CGEND
   ;
statement
   :  (createOp | addInputOp | addDestOp | setValOp)
   ;
createOp
   :  createSym WORD WORD
```

```
   ;
addInputOp
   :  addInputSym WORD NUMBER WORD
   ;
addDestOp
   :  addDestSym WORD WORD NUMBER
   ;
setValOp
   :  ( setValIntOp | setValStringOp | setValDoubleOp
                    | setValCharOp   | setValBoolOp   )
   ;
setValIntOp
   :  setValIntSym WORD NUMBER
   ;
setValCharOp
   :  setValCharSym WORD WORD
   ;
setValBoolOp
   :  setValBoolSym WORD WORD
   ;
setValDoubleOp
   :  setValDoubleSym WORD NUMBER
   ;
setValStringOp
   :  setValStrSym WORD WORD
   ;

createSym:              "CREATE"
   ;
addInputSym:    "ADD_INPUT"
   ;
addDestSym:             "ADD_DESTINATION"
   ;
setValIntSym:   "SET_VALUE_INTEGER"
   ;
setValCharSym:  "SET_VALUE_CHARACTER"
   ;
setValBoolSym:  "SET_VALUE_BOOLEAN"
   ;
setValStrSym:   "SET_VALUE_STRING"
   ;
setValDoubleSym:"SET_VALUE_DOUBLE"
   ;

////////////////// Lexer

class L extends Lexer; // one-or-more letters followed by a newline

CGEND:  '#'
    ;
protected
LETTER: ('a'..'z'|'A'..'Z'|'_'|'.' )
    ;
protected
```

```
DIGIT:  ('0'..'9')
    ;
WORD:   LETTER (LETTER | DIGIT )*
    ;
NUMBER: ('-'|'+')? (DIGIT)+ ('.' (DIGIT)+ )?
    ;
// Whitespace -- ignored
WS      :       (       ' '
                |       '\t'
                |       '\f'
                        // handle newlines
                |       (       options {generateAmbigWarnings=false;}
                        :       "\r\n"  // DOS
                        |       '\r'    // Macintosh
                        |       '\n'    // Unix
                        )
                )+
                { _ttype = Token.SKIP; }
        ;
```

Although the graph in (Fig. 4.1) is a simple graph, it still requires 9 KDLang instructions to build it. To build a corresponding graph using objects and method invocations, like that in (Fig. 2.2) takes more than double this amount. The KDLang language, while suited to easy interpretation by Compeer, is cumbersome when used to develop non-trivial applications. For this purpose, Java code is converted into KDLang, which can then be optimised or executed on Compeer as is. To illustrate the suitability of using Java in place of KDLang for describing object creation and invocation, a code snippet corresponding to (Fig. 2.2) is presented:

```
(new BigInteger("2001")).add(new BigInteger("9"))
```

A detailed description of the process involved in translating Java to an equivalent form of $\mathcal{CG}$s follows. For clarity, images of the corresponding graphs are shown, rather than KDLang examples.

5. Translation Process.

5.1. Motivation. Most single-threaded programs contain a surprising amount of implicit parallelism [10],[11]. This parallelism can be automatically uncovered by converting Java to equivalent $\mathcal{CG}$ graphs, and executing them on the Compeer platform. Using this approach, distributed applications can be produced from single-threaded code without the need for RMI[8] or message passing libraries, such as MPJ[3]. The consequent application deployment is also simpler since the programmer is not responsible for distributing the application across the network or for managing proxy and stub objects.

In contrast, tools such as JavaParty[16] and Do![12] exist which distribute multithreaded Java applications exploiting programmer designed parallelism across a network, in what is known as a "nearly transparent" manner. That is, the tools require that the programmer make only small changes to the application code before it is run. The burden of ensuring synchronisation and concurrency, however, still remains with the programmer.

This work is motivated by the fact that many types of programs lend themselves to the type of conversion described here. These include, parameter sweep applications, key cracks and image processing algorithms.

5.2. ANTLR. ANTLR[15] was used to implement this process because it uses a uniform language for specifying the lexical and parsing rules which make up a grammar. A Java grammar is freely available for ANTLR, which facilitates easy generation of a parser that can recognise all Java constructs.

Action code, in the form of Java snippets, is embedded in a grammar rule wherever an action is desired. For example, to print out the names of all variables in a Java program, a line of code to print out the current token would be added to the rule which recognises variable declarations. To effect translation from Java to KDLang, action code is inserted into relevant grammar rules which outputs the equivalent KDLang to a file.

5.3. Caveats. Note that the Java program to be converted into a $\mathcal{CG}$ should compile and run correctly on a single machine before conversion is attempted, since this translation process performs no type or syntax checking.

Only the class containing the main method is converted. Each of the methods in this class is converted into a $\mathcal{CG}$. The main method is converted into a $\mathcal{CG}$ which uses these other $\mathcal{CG}$s. This gives the programmer control over the grain size of instructions to be distributed. If all involved classes were converted to $\mathcal{CG}$s, the resulting grain size would be so fine as to slow down the computation.

Consider, for example, a program `SortList` which calls a method, `quickSort` in another class, Sorts. If both classes, `SortList` and `Sorts` were converted, then each statement in the `quickSort` method would be translated into nodes in a $\mathcal{CG}$. Many of these statements would be simple comparisons and swaps. It would take longer to schedule the `invoke` nodes representing these operations then it would to carry them out. By compiling the `Sorts` class into a Java class, and converting the main program into a $\mathcal{CG}$, a more useful program is obtained. A call to the `quickSort` method is represented as a single `invoke` node, which when fired executes the `quickSort` method. This method may involve the execution of many Java instructions though it is a single node. Thus the grain size is much coarser than if we compiled all classes.

Better results are obtained by keeping related code together. For example, it would be inefficient to spread nodes controlling a loop across different machines, since they will access each other often. To prevent this, code blocks, such as methods and loop bodies are converted to seperate $\mathcal{CG}$s rather than nodes in a single $\mathcal{CG}$. This facilitates easy distribution of code to machines where it can be unpacked and executed.

5.4. Preprocessing. Before translation begins, the Java code is altered to remove any primitives, and primitive operations, and all arrays are replaced with equivalent calls to the Java Array class. The effect of this is that all variables will be true objects, that is, subclasses of Object, and all actions upon these objects will be carried out via method invocations on those objects. This step is vital since primitive operations are not supported in dynamic invocation in Java.

The first step, removing all primitives, is trivial. Primitive variable declarations e.g., `int num = 7` are rewritten using the relevant wrapper type e.g., `Integer num = new Integer(7)`. This has no effect on methods which require an int but are now passed an Integer, since the Java dynamic invocation mechanism automatically handles this conversion.

All operations on these primitives e.g. `-, ++, <` are converted to method calls to a class written specifically for this purpose. For example, `num < limit` becomes `Operators.lessThan(num, limit)`. The methods in the Operators class are overloaded to handle all primitive types.

The Java Array class is used in place of standard array access. To look up an element in an array, e.g., `list[n]`, a call to `Array.get` is used e.g., `Array.get(list, n)`.

After these modifications are carried out, the code is still valid Java and will execute in the same manner as the original.

Several translation rules are applied to this code to produce an equivalent $\mathcal{CG}$ application.

6. Translation rules.

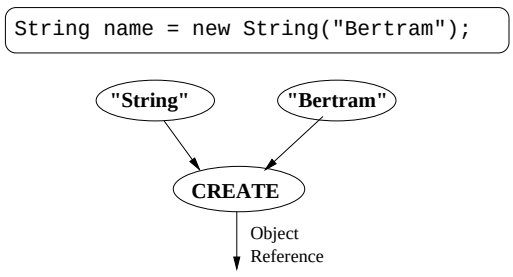

FIG. 6.1. Corresponding Graph for Object Creation

6.1. Object Creation. In $\mathcal{CG}$s, there are no variable names as there are in Java. Relationships such as parameter passing are expressed by linking together nodes. To convert the Java code for objects creation e.g., `ClassName objName=new ClassName(prm);`
only the class name and any parameters are needed, which are used as inputs to a `create` node, as in (Fig. 6.1). The variable `objName` is now associated with this `create` node, and any reference to this variable will be replaced with a reference to this node.

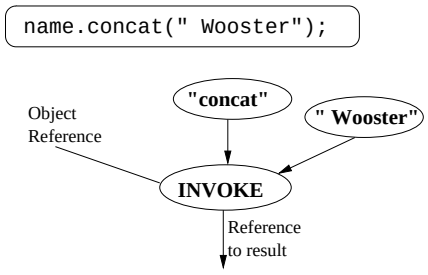

FIG. 6.2. Graph for invocation of a method on an object

6.2. Method Invocation. To invoke a method on an object we need a reference to the object, a method name and, optionally, parameters. The object reference will be an output from a `create` node or a result from another `invoke`. To convert a piece of code of the form:

```
varName = objName.method(param);
```

an `invoke` node is used. The node previously associated with `objName` (from a create or invoke) is used as the object reference. The method name and params make up the remaining inputs, as in (Fig. 6.2). `varName` is now associated with this `invoke` node.

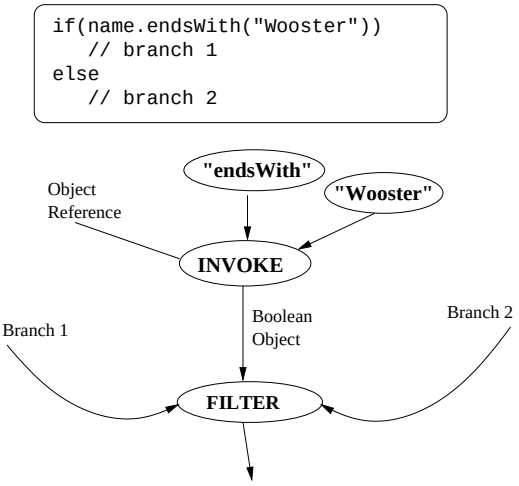

FIG. 6.3. Graph for If-Else statement using Filter node

6.3. If-Else statement. The if-else statement is converted to a Filter node, which takes a boolean parameter and two other nodes, representing the two branches of code to be executed. When the Filter node fires it connects one of the branches to its destination, depending on the truth value of the boolean.

To convert code of the form seen in (Fig. 6.3) a Filter node is added to the graph. If the condition is expressed as a variable e.g., if(property) then the node associated with that variable is added as the boolean input to the Filter node. If the condition, like the branches of code, consists of one or more statements, then those statements are converted to a graph using the translation rules and that graph is used as an input.

The ternary operator is handled identically.

6.4. Loops. In general, with while loops the amount of iterations cannot be calculated at compile time, whereas with for loops it is easily determined. To convert a for loop which results in `n` iterations of some code, we use `spread` node that when fires produces n instances of that code. The `spread` node is a special node that can fire multiple times with one destination.

Caution must be exercised when converting for loops since some are not amenable to this type of parallelisation.

To convert code of the form:

```
for(AnInteger i=0;i.lessThan(100);i.add(1))
    aMethod(i);
```

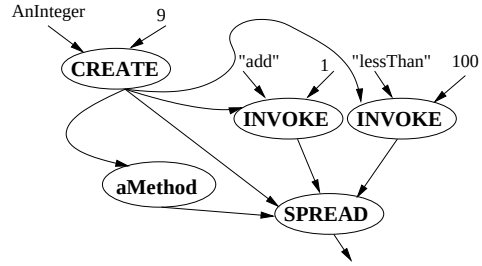

FIG. 6.4. Graph for a For loop

```
for(initCode;conditionCode;incrementorCode)
    mainCode;
```

the initCode, conditionCode and incrementorCode are converted as per the translation rules and added as inputs as in (Fig. 6.4). The initCode can fire immediately since it has a destination. The `spread` node itself can fire too, which causes the conditionCode to execute. If this code returns true then an instance of the mainCode is created. This will continue to happen until the condition becomes false, at which time the `spread` node becomes unfireable. Hence, each time the `spread` node fires, it starts an instance of the mainCode. This is a better approach to starting all N instances at once, since N may be unfeasibly large, or resources may be tight. This method starts as many instances as possible as soon as possible. For a large loop, as the first instances complete execution, more instances can be created as necessary.

An optimised version of the `spread` node takes a mainCode and an integer parameter, N. When fired it creates N instances of the mainCode node and populates each with a unique integer between 0 and N-1.

While loops are handled similarly but only one instance of the code is executed at a time.

```
public String repeat(String str)
{
   return str.concat( str );
}
```

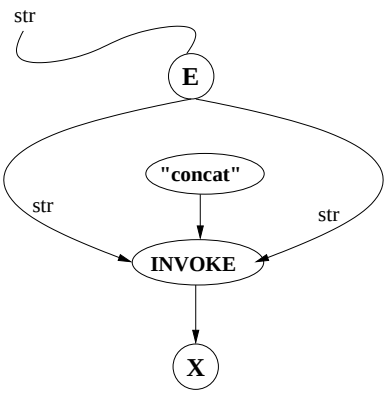

FIG. 6.5. Corresponding Graph for Methods

6.5. Methods. Methods are compiled into $\mathcal{CG}$s in order to keep the related code packaged together for efficient distribution. A $\mathcal{CG}$ has an E (entry) node, which takes any parameters and an X (exit) node, which returns the result.

To convert code of the form:

```
public returnType methodName(params)
{   // code;
```

```
    return result;
}
```

E node is used which takes the supplied parameters as inputs. The method code is converted using the translation steps, with the E node linked to any portions of code which use a parameter. The node associated with the variable being returned is linked to an X node, as in (Fig. 6.5) This completed $\mathcal{CG}$ is given the same name as the Java method including parameter information to prevent ambiguity in cases of method overloading.

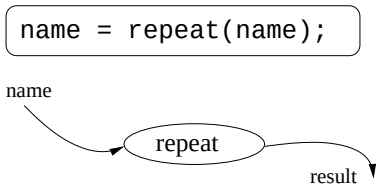

FIG. 6.6. Corresponding Graph for calling a method

To invoke a local method, a node representing the relevant $\mathcal{CG}$ is used, populated with any necessary parameters, as in (Fig. 6.6).

Note the lack of `invoke` node, since this is not an actual method invocation, but a call to another $\mathcal{CG}$.

6.6. Code blocks. Code between curly brackets e.g., {, }, are treated as related code like methods, and are converted into $\mathcal{CG}$s. Any variables accessed in the block are treated as variables and are passed in. The code block is replaced with a call to the $\mathcal{CG}$. This facilitates easy distribution of code e.g., in if statements the two branches could be on seperate machines, but related code is still kept together.

6.7. Synchronising Instructions. Instructions that return values and/or take parameters can be naturally organised into a parallel graph, as in (Fig. 6.7) . In this case it's clear which actions should be taken before other actions can be carried out and so parallelism can be uncovered with no possiblity of executing instruction in the wrong order.

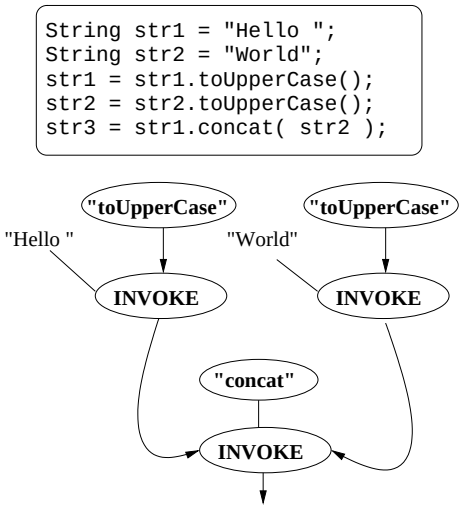

FIG. 6.7. Uncovering Hidden Parallelism

For instructions that return no value, we must ensure instructions are performed in the order the programmer states. To convert code of the form:

```
statement1();
statement2();
```

Converting this code results in two unconnected `invoke` nodes. They must be connected in a manner which guarantees the correct order of execution.

We introduce a `sync` node which takes two parameters corresponding to the statements to be ordered. In (Fig. 6.8) the second statement has no destination so cannot fire. The first statement can fire, and upon

completion a null result is sent to the `sync` node. This fires the `sync` node, which connects it's second parameter, the other statement, to its destination allowing it to fire. This procedure can be used repreatedly to enforce the execution order of any number of instructions.

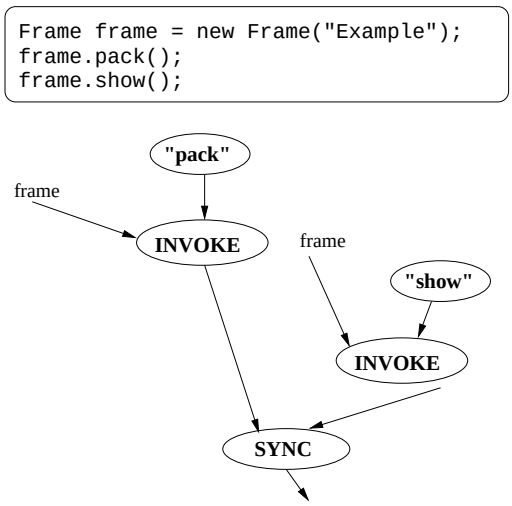

FIG. 6.8. Forcing sequential execution

These translations are enough to enable us to convert most Java programs into $\mathcal{CG}$s.

7. Adding DCOM and CORBA support. Two extra commands, `newDCOM` and `newCORBA`, have been added to this Java grammar to enable support for DCOM and CORBA objects. These are converted just like the Java `new` operator, and method invocations are written just like standard Java invocations. At conversion time, a `create` or `createDCOM` node is produced, depending on the keyword. This also affects the type of invoke node generated later on. Adding this support gives us a very powerful tool that lets us incorporate DCOM and CORBA objects into programs as if they were Java objects.

An example of using a DCOM object called MathsLibrary to test primality of a Java int.

```
DCOM mathsLib = newDCOM MathsLibrary();
int y = 777;
boolean prime = mathsLib.isPrime( y ); // invocation on a DCOM object, assigned
                                       // to a Java primitive
```

A more complex example could interrogate a CORBA object for parameters, use these to create a computation which can be executed across the network and plot the results in, for example, Microsoft Excel.

This method allows a programmer to incorporate any of the many industry applications which supply DCOM or CORBA interfaces into their application. However, since these keywords are not part of the Java language, programs using these features will not compile using a Java compiler. They can only be executed using this system.

8. Sample Applications. The following code fragment produces the graph in (Fig. 8.1). The code is simplified for the purposes of this paper, for example, where print() is used a call to System.out.println should be made, however this would result in the production of create and invoke nodes in the resultant graph. This level of complexity is left to the second example.

```
class CheckSum
{
    void checkNumber(BigInteger n)
    {   BigInteger two = new BigInteger("2");
        print(GetSeqLen(n, 0));
        checkNumber(n.add(two));
    }

    int GetSeqLen(BigInteger m, int i)
```

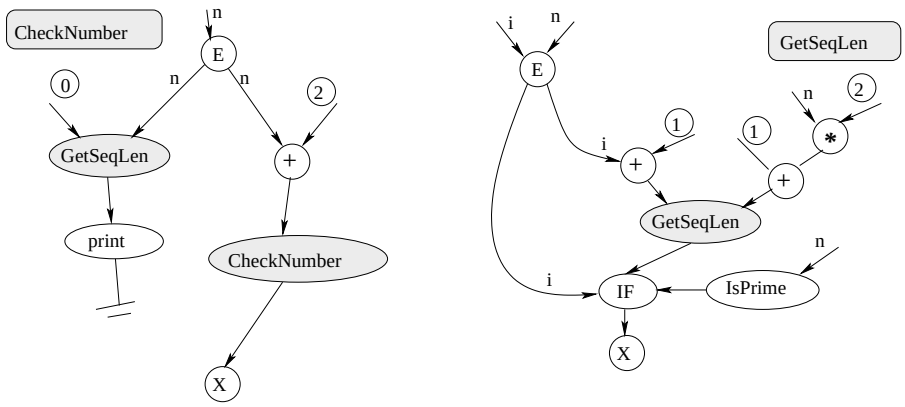

FIG. 8.1. A graph to print the length of sequences of primes generated by taking a prime, doubling it and adding one e.g., 89, 179, 359, 719. The graph is simplified for brevity; isPrime represents an invocation of `isProbablePrime` on the BigInteger N. * and + correspond to the methods `multiply` and `add`. The shaded nodes represent Condensed Graph nodes

```
    {   BigInteger one = new BigInteger("1");

        if(m.isProbablePrime(1000000))
            return GetSeqLen(m.add(m).add(one), i+1);
        else
                return i;
    }
}
```

The example graph in (Fig. 8.1) is built using only standard Java classes. It is a simple recursive graph, used to determine the lengths of sequences of numbers satisfying a certain property. The grain size for this application is too fine to experience a speed up, and actually slows down if run across computers. It is only useful to distribute multiply and addition instructions when the numbers involved are enormous. The application is provided to illustrate the concepts involved in building a $\mathcal{CG}$ from Java objects.

A more realistic example would use at least some custom written code. The application in (Fig. 8.2) uses a class, `MandelLib`, to facilitate calculation of the Mandelbrot Set. The class contains a method `CalcRow` which takes a parameter representing the row to be calculated and returns that row. Parameters such as width, height and magnification are specified at run-time. The graph uses a method, `PlotRow` in another class, `CGWindow` to plot these rows. CGWindow is a general purpose class, used for plotting any data. Rows, rather than individual pixels, are calculated and plotted to increase the grain size of the computation. The application is parallelised using a special node `Spread`, which takes an integer parameter, *I*, and a node *Work* and produces *I* copies of the *Work* node, passing each copy a unique number between 0 and *I-1*. Even though the custom written class, `MandelLib`, contains no network code, the application can still be distributed across the network. The Compeer system handles the transfer of work and results, relieving the programmer of that burden.

An interactive Java Mandelbrot viewer was built using a modified version of this application which returns the data generated rather than displaying it. The viewer loads the KDLang definition of the Mandelbrot graph from a file and replaces three parameters, x, y and magnification, in the graph before submitting it to Compeer. The result is displayed on screen. A user can click on any part of the image causing new coordinates to be generated, and a new job to be submitted to Compeer and subsequently displayed.

This illustrates the ease with which certain types of distributed application can be designed using this technique. Similarly, physical fields, non-linear equations and other operations can be calculated across a network and viewed.

Using DCOM or CORBA these results could be integrated directly into other applications e.g., an MS Access database from which reports could later be extracted.

9. Conclusion and Future Directions. The topics of metacomputing, and decentralised computing, have been the focus of much attention in the recent past. The importance of the field is becoming evident in a variety of application domains. Specific directions include Resilient Overlay Networks [1] and the increase in

```
public void doRow(int row, CGWindow window)
{   MandelLib lib = new MandelLib();
    window.plotRow( lib.calcRow( row ) );
}

public static void main(String [] args)
{   CGWindow window = new CGWindow();

    for(int i=0;i<100;i++)
        doRow( i, window );
}
```

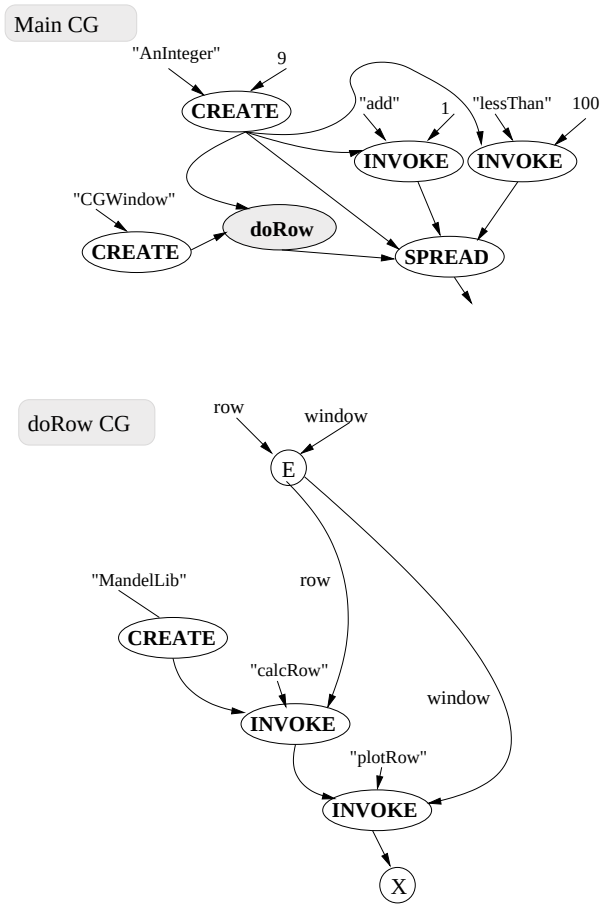

FIG. 8.2. Example program snippet to calculate and display the Mandelbrot Set converted to an automatically parallel $\mathcal{CG}$ The shaded nodes represent Condensed Graph nodes

grid based projects such as Globus.

This paper addresses issues of usability and accessibility. The approach is to use a familiar high-level language and map it onto a dynamically, easily deployed architecture which can by dynamically constructed to execute the task. Novel features include it's exploitation of the $\mathcal{CG}$ computation model to facilitate dynamic load balancing through a process of reflection and applications whose behaviour can be customised at run time.

While the system, as presented here, has obvious benefits in that it reduces the cost in both setting up a metacomputing platform and developing parallel distributed applications, there is much work to be done. For example, the Compeer platform itself needs to be comprehensively benchmarked and compared with other technologies. Methods for safely uncovering more parallelism during the translation process must be investigated and used to maximize the the benefits of distributing the computation. Also, some deployments, particularly

across the Internet, will not become viable until security is properly addressed.

It is envisaged that when these steps are completed, it will be possible for internet based cooperative metacomputers to be constructed in an ad hoc fashion, allowing users to easily share data and services, and create applications to exploit these.

Acknowledgments. The support of the Informatics Research Initiative of Enterprise Ireland is gratefully acknowledged.

REFERENCES

[1] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris, *Resilient overlay networks*, in Symposium on Operating Systems Principles, 2001, pp. 131–145.
[2] F. Berman and R. Wolski, *The apples project: A status report*, 1997.
[3] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox, *MPJ: MPI-like message passing for Java*, Concurrency: Practice and Experience, 12 (2000), pp. 1019–1038.
[4] distributed.net. `http://www.distributed.net` 2002.
[5] I. Foster and C. Kesselman, *Globus: A metacomputing infrastructure toolkit*, The International Journal of Supercomputer Applications and High Performance Computing, 11 (1997), pp. 115–128.
[6] J. Gosling et al., *The Java Language Specification*, GOTOP Information Inc., 5F, No.7, Lane 50, Sec.3 Nan Kang Road Taipei, Taiwan; Unit 1905, Metro Plaza Tower 2, No. 223 Hing Fong Road, Kwai Chung, N.T., Hong Kong, 19xx.
[7] R. Henderson and D. Tweten, *Portable batch system: External reference specification*, tech. report, NASA Ames Research Center, 1996.
[8] JavaSoft, *Java remote method invocation—distributed computing for Java*, white paper, Sun Microsystems, Inc., 1998.
[9] R. Karp and R. Miller, *Properties of a model for parallel computations: Determinacy, termination, queuing.*, SIAM Journal of Applied Mathematics, 14 (1966), pp. 1390–1411.
[10] B. C. Kuszmaul, *Suif to dataflow*. Second SUIF Compiler Workshop, Stanford University, August 1997.
[11] M. S. Lam and R. P. Wilson, *Limits of control flow on parallelism*, in Nineteenth International Symposium on Computer Architecture, Gold Coast, Australia, 1992, ACM and IEEE Computer Society, pp. 46–57.
[12] P. Launay and J.-L. Pazat, *A framework for parallel programming in JAVA*, in HPCN Europe, 1998, pp. 628–637.
[13] M. J. Lewis and A. Grimshaw, *The core legion object model*, Tech. Report CS-95-35, 1995.
[14] D. A. P. . J. P. Morrison, *Nectere: A general purpose environment for computing on clusters, grids and the internet*, in International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '02), Las Vegas, USA, June 2002, pp. 719–726.
[15] T. Parr and R. Quong, *Antlr: A predicatedll (k) parser generator*, Journal of Software Practice and Experience, 25 (1995), pp. 789–810.
[16] M. Philippsen and M. Zenger, *JavaParty — transparent remote objects in Java*, Concurrency: Practice and Experience, 9 (1997), pp. 1225–1242.
[17] L. F. Sarmenta, *Sabotage-tolerance mechanisms for volunteer computing systems*, in 1st International Symposium on Cluster Computing and the Grid, Brisbane, Australia, May 2001, p. 337.
[18] B. Schneier, *Applied Cryptography*, John Wiley And Sons, 2nd ed., 1996.
[19] SETI@Home. http://setiathome.ssl.berkerly.edu, 2002.
[20] A. Silis and K. A. Hawick, *The DISCWorld Peer-To-Peer Architecture*, in Proc. of the 5th IDEA Workshop, Fremantle, 1998.
[21] G. software GmbH. `http://www.genias.de/genias/english/codine.html` September 1995.
[22] R. Wolski, J. Brevik, C. Krintz, G. Obertelli, N. Spring, and A. Su, *Running EveryWare on the computational grid*, 1999.

Edited by: Dan Grigoras, John P. Morrison, Marcin Paprzycki
Received: October 01, 2002
Accepted: December 15, 2002