A Concurrent and Distributed Model for Complex Boolean Calculations

Steinbach, Bernd
Freiberg University of Mining and Technology
Institute of Computer Science
steinb@informatik.tu-freiberg.de

Dobrev, Tochko
Freiberg University of Mining and Technology
Graduate School of Spatial Statistics
dobrev@merkur.hrz.tu-freiberg.de

Abstract
This paper describes the architecture of a heterogeneous, concurrent, and distributed system, which can be used for solving large computational problems. At first we introduce the system architecture and the techniques used to implement it. To estimate the performance of the distributed system we compute a boolean task parallelly.

1 Introduction
Problems from many scientific and economic fields need a large computer power for their solution. The numeric simulation of complex systems like weather forecast, climate modelling, molecular biology and circuit design are some of such problems. There are two approaches to solve them. Either an expensive parallel supercomputer has to be used [1], or the computer power of workstations in a net can be bundled to compute the task distributed [2]. The second approach has the advantage that we use the available hardware cost-effectiv.

Every academic or enterprise network can be treated as a virtual parallel computer on which concurrent and distributed applications can be implemented. Such an application is a program which parts are distributed on different machines and can be computed independent as they exchange messages with each other over the network by suitable communication mechanisms [3].

In this paper we present the architecture of a concurrent and distributed system and estimate its performance by computing a boolean task.

2 Distributed calculation model
2.1 Basic technologies
The system was developed with the objectoriented hardwareindependent programming language Java\textsuperscript{TM} [4] and integrated techniques RMI\textsuperscript{1} and JNI\textsuperscript{2}.

Java\textsuperscript{TM} RMI\textsuperscript{3} enables the programmer to create distributed Java-to-Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines (JVM) [6] on different hosts. A Java program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in the bootstrap naming service called rmi\textsuperscript{4}registry\textsuperscript{5} provided by RMI or by receiving the reference as an argument or a return value. A

\textsuperscript{1}Java\textsuperscript{TM} Remote Method Invocation.
\textsuperscript{2}Java\textsuperscript{TM} Native Interface.
\textsuperscript{3}Each computer has such a register, in which its remote objects are administered by unique names.
client can call a remote object in a server, and this server can also be a client of other remote objects.

The Java™ RMI [7] is a native programming interface that allows Java code that runs inside a JVM to interoperate with applications and libraries written in C/C++ and Assembler programming languages. Thus, it is possible to integrate such libraries in the distributed system and to use it for parallel computation of different tasks.

2.2 Remote Method Invocation

Like any other application, a distributed application built using Java™ RMI is made up of interfaces and classes. The interfaces define methods, and the classes implement the methods defined in the interfaces and, perhaps, define additional methods as well. In a distributed application some of the implementations are assumed to reside in different virtual machines. Classes that implement methods that can be called across virtual machines have remote objects as instances.

Every class that want to instance remote objects has to implement direct or indirect the interface java.rmi.Remote and has to extend the class java.rmi.UnicastRemoteObject from the Java standard programming library. After compiling such a class, for example MYCLASS.java, with javac it has to be compiled with rmic. The rmic command decomposes the class structure and produces as output two other classes MYCLASS.Stub.class and MYCLASS.Skel.class called Stub and Skel components for any object of MYCLASS.class.

When invoking a method on remote object the Stub is downloaded on the caller machine and acts as client’s local representative or proxy for the object. The caller invokes a method on the Stub which is responsible for carrying the method call on the remote object as building the connection to the Skel and transmitting the parameters to the remote JVM. The Skel takes the parameters and runs the method (see Figure 1). Thereby the caller has the impression that all happens local.

2.3 Central object management

In the default case RMI implements one local register per computer that has remote objects. This is quite unfavorable because the distributed system has to check every register to find out which objects are available. Therefore there is a big overhead at starting time, where all objects have to be referenced parallelly.

To avoid this we implement one central register which manages the remote objects on all computers (see Figure 2) and design it as remote object, too.

The class CentralReg that realizes the central register implements the interface java.rmi.Remote indirectly and extends the class java.rmi.UnicastRemoteObject (see Figure 3).

---

4The Java™ compiler.

5The RMI compiler.
In the constructor of the class CentralReg the methods createRegistry(), getRegistry() and rebind() from the packages java.rmi.registry and java.rmi have been used to create the default local rmiregistry and to bind in it under unique name name the current instance of the class that implements the central register. The other computers can reference the register and bind there its remote objects:

public CentralReg(String name) throws RemoteException, AccessException {
    this.name = name;
    LocateRegistry.createRegistry();
    this.register = LocateRegistry.getRegistry();
    this.register.rebind(name, this);
}

By invoking the method bindInRegistry(String name, Remote object) on the current instance of the class CentralReg it is possible to bind a remote object into the central register. The Stub component is downloaded on the caller machine and realizes the binding process:

public void bindInRegistry(String name, Remote object, String password) throws RemoteException, AccessException {
    if( !this.magic_password.equals(password) )
        throw new AccessException("incorrect password");
    this.register.rebind(name, object);
}

### 2.4 Design of the remote objects

The parts of complex computational tasks have to be distributed and send from one central computer to other machines. They solve these as they work with methods from native libraries and return the partial results to the central one. The distributed system must have a service that is able to realize this approach.

The objects of the class RemoteClassImpl are implemented as remote and its methods can invoke the methods of the class NativeMethods that accesses native libraries (see Figure 4).

All the instances of RemoteClassImpl have to be activated on each computer that is used for the distributed work. This is done by binding the objects into the central register:

```java
public class Registering { ...
    public static SimpleNaming registry = null;
    public static RemoteClassImpl robject = null;
    ...
    public static void main(String[] args) {
        try {
            String ObjectName = "object1";
            String RegistryName = "CentralRegistry";
            String lookfor = "/13x.2x.1xx/" + RegistryName;
```
registry = (SimpleNaming)Naming.lookup(lookfor);
robject = new RemoteClassImpl();
registry.bindInRegistry(ClassName, robject, password);
System.out.println(ClassName + " was successfully bounded!");
} catch { ... }

The string variable lookfor contains information where the central register can be found. The
method lookup(lookfor) from the class java.rmi.Namingchecks this information and if it finds
the remote object it downloads his Stub. The current object robject has to be created and bound
under unique name ClassName into the central register.

2.5 Access to the native libraries

The Java™NI accesses native libraries written in C/C++ or Assembler programming language.
To describe how the distributed system works with such libraries an example is given.

The local host give the task to a remote host to generate and orthogonalize a TVL6 and to return to
it the result. The library for logic design XBOOLE [8, 9] is used for generating and orthogonalizing
the TVL. The methods of this class are declared as native that means they have to be implemented
with a native language.

    public class NativeMethods {
        public native int[] generateTVL(int Vars); ...
    }

After compiling with javac a javah utility program is used to generate a header .h file from the
NativeMethods class. The header file provides a C function signature for the implementation of
the native method int[] generateTVL(int Vars) defined in this class. Now, the implementation
for this method has to be done in another language than Java:

    #include <stdio.h>
    #include <malloc.h>
    #include "NativeMethods.h"
    #include "xb_port.h"

    JNIEXPORT intArray JNICALL Java_NativeMethods_generateTVL
    ( JNINativeEnv *env, jobject self, jint Vars ) {
        uns *space, *list, *list_help, *x01, *buf,
        vmax, typ, lengt, i, j, index;
        space = list = list_help = NULL; vmax = 32; index=0;
        char *variable[] = {"x01", "x02", ...,"x32"};
        MAKE_SD(&space, vmax);
        PUT_SL(i, &space);
        MAKE_TVL(&space, "x01", &list);
        for(i=1; i<=Vars; i++) {
            MAKE_TVL(&space, variable[i-1], &list_help);
            PEL(&list, i, &list_help, ONE, &list); }
        SFORM(&list, A_FORM, &list);
        ORTH(&list, &list);
        typ = GET_TYP(&list);
        intArray TVL = env->NewIntArray(typ*length);
        jint* body = env->GetIntArrayElements(TVL, 0);
        buf = (uns*)malloc(typ*sizeof(uns));
        FIRST_ROW(&list, i, buf);
        for( j=0; j<typ; j++ ) {

6Ternary vector list.
body[index++] = *(buf+); }
while( NEXT_ROW() ) {
    for( j=0; j<typ; j++ ) {
        body[index++] = *(buf+); }
return TVL; }

The C Program generates (MAKE_TVL()) and orthogonalizes (ORTH()) the TVL and finally stores it in a java array. To use the implemented program with the distributed system it has to be compiled as shared library\footnote{.dll for MS Windows or .so for UNIX and Linux.}. The library is loaded from the java distributed application:

\[
\text{static \{ System.loadLibrary("NativeLibrary"); \}}
\]

The example can be solved as the local host references a remote object from the central register, downloads its Stub and invokes the method int[] Generate(int Vars) on it. This method invokes the native one int[] generateTVL(int Vars) on the remote host where is the \textit{XBOOLE} library:

\[
\text{public class TVL \{ ...}
\text{    RemoteClass object1 = null; ...}
\text{    public static void main(String[] args) \{ ...}
\text{        try {}
\text{            lookfor = "/13x.2x.1xx/CentralRegistry/object1";}
\text{            object1 = (RemoteClass)Naming.lookup(lookfor);}
\text{            int[] TVL = object1.Generate(int Vars);}
\text{            System.out.println("TVL was generated successfully");}
\text{        } catch(...) \{\}
\text{    \}}
\text{}}
\]

2.6 Parallel work

To compute a task parallely with the distributed system the \textit{Java\textsuperscript{TM} Threads} [10] mechanism to reference remote objects simultaneously in the central register was used. The presented start up model is used for the parallel computation of a Boolean equation with \textit{Shannon decomposition} (see paragraph 3):

\[
\text{public class MainStart \{ ...}
\text{    public static void main(String[] args) \{ ...}
\text{        int n=2;}
\text{        int ObjectID=0;}
\text{        String Registry = "/13x.2x.1xx/CentralRegistry";}
\text{        Thread[] worker = new Thread[n];}
\text{        for(i=0; i<n; i++) {
\text{            worker[i] = new Thread(new Starting(Registry, ObjectID));
\text{            worker[i].start();}
\text{            objectID++; \}} \}
\text{}}
\]

The Threads activate the work on two computers. The class \textit{Starting} implements the interface \textit{Runnable}\footnote{Java standard library, package \texttt{java.lang}.} and in its \textit{run()} method. The object which is identified with \textit{ObjectID} has to be referenced.

3 Decomposition of Boolean task

The boolean equation

\[
\bigoplus_{i=1}^{n} x_i = 1
\]
can be solved by Shannon decomposition

\[
\bigoplus_{i=1}^{n} x_i = \mathbf{T}_1 \cdot \big( \bigoplus_{i=1}^{n} x_i \big) \bigoplus_{i=2}^{n} x_i \cdot \big( 1 \bigoplus_{i=2}^{n} x_i \big)
\]

\[
\text{Task 1} \\ \text{Task 2}
\]

For few variables formula (2) can be calculated sequentially. Up 20 variables the sections (Task 1 = 1) and (Task 2 = 1) are to be processed parallel and on different computers, in order to reduce the computing time.

4 Results

The used hardware consists of a PC Intel 2xPentium II 450 MHz, 256 MB RAM with SuSE Linux 6.3 and a PC Intel Pentium III 733 MHz, 256 MB RAM with MS Windows Workstation 4.0 connected by TCP/IP Ethernet.

To estimate the performance of the distributed system the time for the computation of the Boolean equation solved by Shannon decomposition has to be measured in the sequential and parallel case. The results were examined only within the area 15-23 variables (see Table 1).

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>variables</td>
<td>time [seconds]</td>
</tr>
<tr>
<td>15</td>
<td>0.02</td>
</tr>
<tr>
<td>16</td>
<td>0.06</td>
</tr>
<tr>
<td>17</td>
<td>0.26</td>
</tr>
<tr>
<td>18</td>
<td>1.49</td>
</tr>
<tr>
<td>19</td>
<td>7.36</td>
</tr>
<tr>
<td>20</td>
<td>39.50</td>
</tr>
<tr>
<td>21</td>
<td>168.02</td>
</tr>
<tr>
<td>22</td>
<td>676.0</td>
</tr>
<tr>
<td>23</td>
<td>2768.25</td>
</tr>
</tbody>
</table>

Table 1: Results of the Shannon decomposition

Under 16 variables the parallel computation is slower as the sequential due to the network transfer. From 16 the parallel case is faster and the time difference between the two cases grows up exponentially (see Figure 5).

Figure 5: Shannon decomposition a) variables/time[hours]; b) variables/time[minutes];
5 Conclusions

The joint computing on a set of computers connected by a network is not just a subfield of parallel computing. Distributed computing is deeply concerned with problems such as reliability, security and heterogeneity that are generally regarded as tangential in parallel computing.

The presented distributed system can be a useful tool for the solving of different complex computational tasks. Due to its heterogeneity the system can use the processor power of each computer with a running JVM. The interface SimpleNaming and the classes RemoteClassImpl and NativeMethods have to be copied on the current machine and the RemoteClassImpl has to be instanced by an appropriate service (2.4) that binds class objects in the central register and enables the computer to be ready for some work. The class NativeMethods realizes an interface between Java™ and the native programming level. Thus, libraries such as XBOOLE written in C/C++ or Assembler can be used for parallel computations.

The partitioning of the tasks is based on rough-grained decomposition. The class MainStart implements a parallel start up mechanism (2.6) that identifies every computer on which one part of a computational problem has to be solved. Thus, it would be possible to handle large pieces of a distributed task and to exploit the power of the involved processors fully.

References