A contribution towards a Distributed Java Virtual Machine

João Fernando Ferreira
joaoferreira@di.uminho.pt

Technical Report
2005, November

PPC-VM
Portable Parallel Computing based on Virtual Machines
(Project POSI/CHS/47158/2002)

Departamento de Informática da Universidade do Minho
Campus de Gualtar – Braga – Portugal

Abstract

The work described in this report is part of *PPC-VM* project[1] (Portable Parallel Computing based on Virtual Machines). The *PPC-VM* project aims to build an environment to support the development and execution of parallel applications that efficiently execute on a wide range of computing platforms, based on virtual machines.

This work contributes to *PPC-VM* project with a parallel computing environment, which aims to simplify the implementation of parallel applications and to test the paradigms/methodologies developed within the *PPC-VM* project.

The parallel computing environment uses the Java programming language and it provides two components: a skeleton catalog implemented as an abstract class library and an automatic object distribution platform, based on source code generation. The former helps programmers creating parallel applications, while the latter transparently distributes objects in a parallel distributed environment.

Application area: Parallel computing

Keywords: parallel computing, parallel programming, automatic object distribution, skeletons, Java

[1] POSI/CHS/47158/2002

Resumo

O trabalho descrito neste relatório está integrado no projecto *PPC-VM* [2] (Portable Parallel Computing based on Virtual Machines). O projecto *PPC-VM* tem como objectivo a construção de um ambiente que suporte o desenvolvimento e a execução de aplicações paralelas, de forma eficiente numa vasta gama de sistemas baseados em máquinas virtuais.

Este trabalho contribui para o projecto *PPC-VM* com um ambiente de computação paralela que pretende simplificar a implementação de aplicações paralelas e testar paradigmas/metodologias desenvolvidas no contexto do projecto.

O ambiente de computação paralela desenvolvido suporta aplicações em Java e fornece dois componentes: um catálogo de esqueletos (*skeletons*) que é implementado como um conjunto de classes abstractas, e uma plataforma de apoio à execução que realiza a distribuição automática de objectos, baseada na geração de código-fonte. Enquanto que a primeira componente é utilizada para ajuda na criação de aplicações paralelas, a segunda é utilizada para distribuir transparentemente objectos num ambiente computacional paralelo ou distribuído.

Área de Aplicação: Computação paralela

Palavras-Chave: computação paralela, programação paralela, distribuição automática de objectos, esqueletos, Java

[2] POSI/CHS/47158/2002

Contents

List of Figures

List of Tables

1. Introduction

1.1. Context

The work presented in this report was integrated within the *PPC-VM*[3] (Portable Parallel Computing based on Virtual Machines) project (POSI/CHS/47158/2002), and developed at the *Grupo de Engenharia de Computadores - Departamento de Informática* under the supervision of *Dr. João Luís Sobral* and *Professor Alberto José Proença*.

The *PPC-VM* project aims to build an environment to support the development and execution of parallel applications that efficiently execute on a wide range of shared computing platforms, based on virtual machines (in particular, on the Java virtual machine). A virtual machine is a piece of computer software that isolates the user application from the computing platform. Any application compiled for a virtual machine can be executed on any computer platform, instead of having to produce separate binary versions of the application for each computer and operating system. The application is run on the computer either by interpreting the code (the original or an intermediate level) or through Just In Time (JIT) compilation.

The *PPC-VM* project description, taken from the submission form, is as follows:

> "PPC-VM *project aims the research of methodologies and tools to help the development of scalable parallel applications that can take advantage of a large number and variety of shared computer resources. The main focus is on the development of methodologies to support efficient fine-grained parallelism (object oriented, specified by fine-grained active objects), whose grain-size can be dynamically adjusted to efficiently use the available resources, matching the available computing and communication bandwidth. This includes the dynamic determination of the number of computer resources that can be efficiently used by the application on particular running conditions. The research will follow a virtual machine based approach, since it provides application code compatibility, supporting dynamically downloaded code and can transparently provide additional services. Additionally, virtual machines are a strong trend in the programming community.*
>
> *The key research issues on this project aim to provide:*
>
> - *High-level specification of scalable parallel applications, supporting fine-grained tasks based on active objects that can be efficiently executed on a wide range of computing resources, including reconfigurable hardware. This includes the efficient mapping of high level scalable parallel programs to virtual machine level;*

[3] http://gec.di.uminho.pt/ppc-vm

- *Parallelism extraction from source or intermediate code, compile time estimation of object granularity information and inclusion of inter-objects dependencies information into object assemblies; the obtained information increases application parallelism and improves the efficiency of the run-time decision making.*
- *Load distribution and granularity control as a virtual machine service, providing transparent and efficient use of a wide range of shared and heterogeneous computing resources.*

The resulting methodologies are implemented either by extending a virtual machine or by building a new layer on top of an existing virtual machine. This implementation will provide several new services, such as dynamic load distribution and granularity control, and tools that map high-level parallel applications to this environment. "

This work contributes to *PPC-VM* project with the specification and implementation of a simple parallel computing environment, which is meant to simplify the design and the implementation of parallel applications. The next section presents this environment, explaining the interaction between its components and introducing some key concepts that will be mentioned throughout the following chapters.

1.2. An integrated approach to parallel computing using Java

1.2.1. Specification of a parallel computing environment

The parallel computing environment described in this work relies on two key concepts: a **skeleton catalog** and **automatic runtime object distribution** based on source code generation.

The skeleton catalog is a collection of code templates implemented as an abstract class library. The catalog is supplied to the programmer to help him/her to create Java code for a parallel/distributed computing platform. Skeletons are abstractions modelling a common, reusable parallelism exploitation pattern [ADT03, Col04]. A skeleton may also be seen as a high order construct (i.e. parameterized by other pieces of code) which defines a particular parallel behaviour.

Distribution of objects among computing nodes (either in a distributed environment or in a parallel cluster) can be statically performed, if it is performed at compile time, or dynamically, if there is a runtime system that decides where to create the objects; it can also be explicitly declared by the user, or it may occur without human intervention. To implement an automatic runtime object distribution, several alternative ways can be used; some require extensions

to the programming language, others may simply achieve it through a tool that generates adequate source code.

The parallel environment provides a skeleton-based framework, which can be used to structure parallel applications. A framework is a support structure in which another software project can be organized and developed. Programmers only have to opt for the appropriate skeleton and define the parameters (i.e. pieces of domain-specific code).

An application developed according to this skeleton-based framework is ready to be transparently distributed among the available resources, using another component from the environment: a source code generator.

This environment is different from other research environments in the way that it uses different and independent components for distinct tasks; it uses the skeleton-based framework to structure parallel applications and it uses the source code generator to support dynamic objects distribution.

The independence between these two components brings some advantages:

- within the skeleton-based framework, programmers can develop a structured application and run it in a non-distributed environment; this allows the programmer to test the application before running it in a distributed environment;
- programmers can use the source code generator with common applications that are not structured using the skeleton-based framework;
- programmers can replace the skeleton-based framework by other frameworks or libraries and maintain the source code generation; this allows the use of alternative ways to structure parallel applications.

1.2.2. Implementation

The main contribution of this work is a Java implementation of the parallel computing environment described above.

The skeleton catalog was implemented as a set of Java abstract classes. To write parallel applications, programmers must express their structure using the available skeletons and define the skeletons parameters, refining the desired abstract classes and writing the domain-specific code.

Object distribution is achieved through a tool that transforms the original Java source code and introduces new support classes. Objects are distributed at runtime. Thus, the environment provides an automatic runtime object distribution, based on source code generation.

This work also contributes with an analysis and description of the Java Grande Forum[4] parallel algorithms, stressing distribution issues. These algorithms are useful to test and validate the environment.

Issues

The first priority of the work described in this report was to allow automatic runtime object distribution. The first approach to generate code did not concern about details like static variables, direct access to instance variables and class inheritance. After the first approach to automatic runtime object distribution, the priority was to simplify the programmer's task. We decided to explore the skeleton concept and we built a skeleton-based framework based on class inheritance.

Full implementation of the parallel computing environment required the application of the object distribution tools to structured code with the skeleton-based framework. This work addressed a partial implementation of the environnment and associated tests and evaluation, leaving for future work the remaining implementation.

1.3. Content overview

Chapter 2 describes an approach to distribute objects and a corresponding Java implementation. It shows some examples, presents some limitations and briefly describes a C# implementation.

Chapter 3 describes a skeleton-based Java framework built to help programmers to structure their parallel applications. It also introduces the skeletal approach to parallel programming by describing what a skeleton is and by presenting some common skeletons.

Chapter 4 presents an evaluation methodology complemented with a comparative study of alternative implementation strategies to distribute objects and a comparison between two skeletons from the skeleton-based framework.

Chapter 5 concludes this report, with a discussion on the obtained results and some conclusions about the work done. It also suggests some improvements to the work.

Appendix A documents the Java Grande Forum parallel algorithms.

Appendix B presents documents related with the automatic object distribution implementation: a brief analysis of existing parser generators (and related tools) and the description of the frontend script used to simplify the source code transformation and the automatic object distribution.

[4] http://www.epcc.ed.ac.uk/javagrande/javag.html

2. Code generation for distributed computing

2.1. An approach to distribute objects

Objects interact with each other when a service is requested - a client object, $c1$, calls a method from a server object, $s1$ - and/or a reply is given - a method in the server $s1$ returns a value to c1. Figure 1 illustrates this interaction; an object may also place requests on several other objects, as shown in Figure 2.

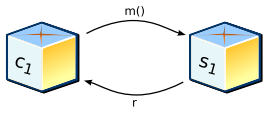


Figure 1: Object c_1 requests service to object s_1

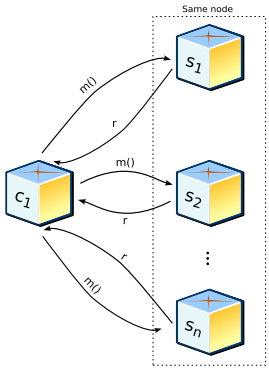


Figure 2: Object c_1 requests service to objects s_1, s_2, ..., s_n

In a distributed environment, where objects may be placed across several computers, or computing nodes in a cluster, a mechanism is required to support transparent distribution of objects and their corresponding communication channels. A programmer expects that objects will be automatically distributed without his/her direct intervention, assuming this job to be part of the operating execution environment. The work developed here had this in mind and considered it as a goal.

To support transparent and efficient object distribution, additional concepts need to be presented:

- a client object always expect the server object to be in the same process, when it calls a method; to provide this facility when the server object is remotely placed, the server object in the same process is replaced by another that mimics its interface and acts as it was the server object; we call this the **proxy object**;

- the server object needs to be remotely created - we call it the **implementation object** - and an entity in the remote node must be able to create these new objects, when requested to do so; we call this entity an **objects' factory**;
- the location of objects in remote nodes may be statically defined at compile time, or may be dynamically placed in run time; this tuning requires an entity to dynamically decide where to place the remote object; we call this load manager in our current project the **cluster manager**. Figure 3 illustrates how these concepts interact.

One critical issue in efficient object distribution, where several computing nodes are competing candidates to host the implementation object, is the strategy to make a decision where to place the implementation object. This decision may be complex and dependent from many parameters: memory usage, CPU usage, number of processes, etc.

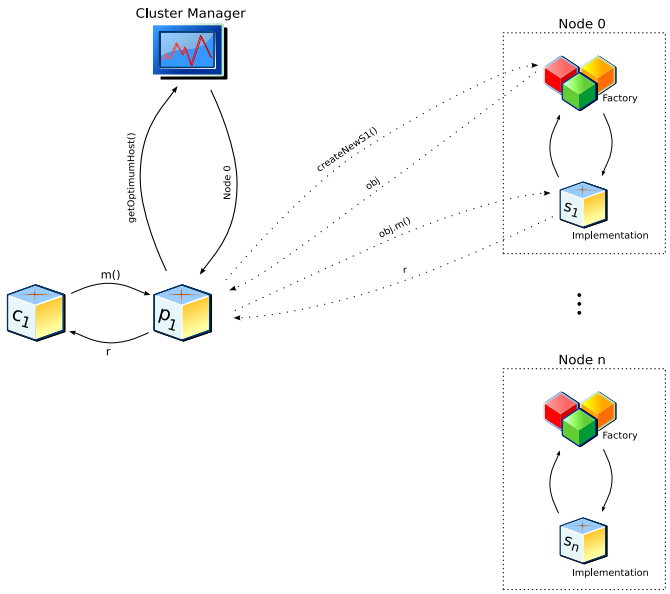


Figure 3: Automatic object distribution: interaction between the components

In this distributed environment, s_1 object is replaced by a proxy object (p_1) and c_1 calls the proxy m method. The decision where to place the implementation object is taken when the proxy object is created. This decision is taken in two steps:

- the proxy asks the cluster manager what is the optimum host to create a remote object (`getOptimumHost()`);
- the proxy places a request to the factory at that host (in this example it is `Node 0`) to create a remote object (`createNewS1()`); the factory returns a reference to the new remote object (`obj`).

Once the remote objects are created, a service request follows these steps:

- client c_1 calls the proxy method $m()$;
- the proxy calls a method on the new remote object, using the reference returned by the factory (`obj.m()`);
- the remote object execute method m and *r*eturns a value to the proxy;
- the proxy *r*eturns the value to the client object c_1.

2.2. A Java implementation

To implement an automatic distribution of objects, several alternative ways can be used; some require extensions to the programming language, others may simply achieve it through support classes. The latter was the adopted approach to implement automatic object distribution, which supports the execution of Java code in this project. A parser transforms the original source code and introduces new classes to represent proxies and the objects' factories.

2.2.1. Parser generator

Two approaches were followed to interpret and transform Java source code: either to adopt an existing parser, or to create a new one from scratch. The former requires a careful analysis of available tools, while the latter may lead to faster implementations. We have chosen the first one because it reduces substantially the development time of this stage.

Appendix B.1 contains a short description of the performed analysis to several parser generators. We opted for the *JParse* library, based on *ANTLR*, since it already has a Java grammar which produces an abstract syntax tree (AST). This feature is relevant to help to reduce the development time and to simplify the code generation.

2.2.2. Code generation

The chosen tool to transform source code creates an AST and it has a tree parser to traverse that AST. Thus, the code generation strategy is as follows:

- $\mathbb{S}$ class source code is parsed and an AST representing that class is created;
- the created AST is traversed several times and in each of these traverses:

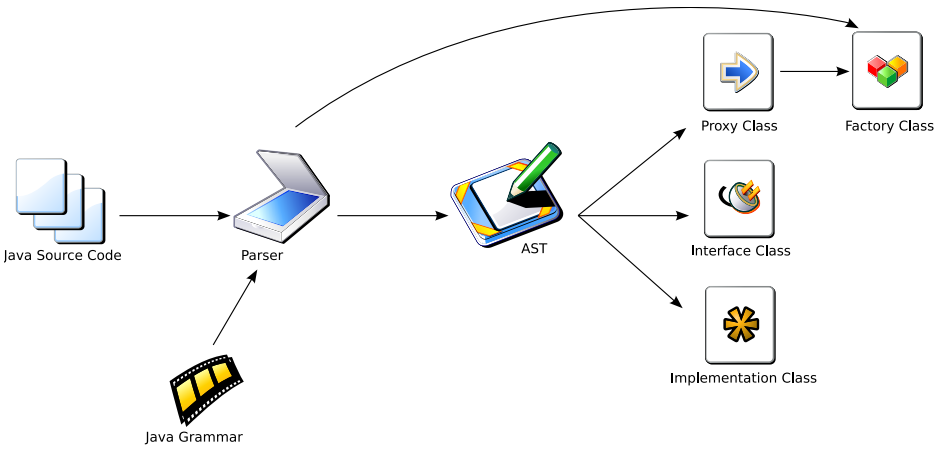


Figure 4: Class generation strategy

  - a new class with the same name and with the same interface is created (**proxy**); the generated proxy constructors create a remote object that is used by all the other methods to redirect the method calls;
  - a new interface file named `I`$\mathbb{S}$ is created; this interface represents the original class public methods;
  - a new class named $\mathbb{S}$`Impl` is created (**implementation**); this class implements the interface `I`$\mathbb{S}$, i.e., it implements the original methods' code; this class is a subclass of `RemoteObject`, which means that its instances are remote objects;

- a class named `PPCFactory` (**factory**) and its interface are created; this class defines methods that create implementation objects (they are generated from the original constructors); this class is a subclass of `RemoteObject`.

This strategy is illustrated in figure 4.

Besides the generated classes, there is a `PPCClusterManager` class, which takes the decisions where to create the remote objects: the current prototype uses a round-robin strategy.

The described approach depends on remote method invocations and on the definition of remote objects. The Java language offers the Java Remote Method Invocation (Java RMI) mechanism, which enables the programmer to invoke a method on a remote object. There are two popular forms of RMI: the pure Java RMI and RMI-IIOP (RMI over the Internet Inter-ORB Protocol). The difference between the two is that RMI-IIOP is compatible with CORBA, since it uses the IIOP protocol of CORBA as the underlying protocol for RMI communication.

The generated code is based on the RMI-IIOP form.

Remote Method Invocation (RMI)

There are three processes involved in a remote method invocation:

- a *client*, which invokes the remote method;
- a *server*, which owns the remote method;
- a *name service*, which allows to register remote objects with a name and returns references to remote objects; since both the client and the server may reside on different address spaces, a mechanism is required to connect them; the name service provides this connection.

The steps described in page 7 omit the queries to the *name service*. However, the *name service* is used to get the references to remote factories, which are registered with the host name where they are running.

Any object can be passed as an argument or returned as a value to or from a remote method as long as it is a primitive data type, a remote object or a serializable object (if it implements the interface `java.io.Serializable`). One critical issue in efficient object distribution is whether an object is passed by reference or by value. Using RMI, arguments and return values are passed as follows:

- remote objects are passed by reference. A remote object reference is a *stub*, which is a client-side proxy that implements the complete set of remote interfaces that the remote object implements.
- local objects are passed by value (a copy of the object), using object serialization. By default all fields are copied, except those that are marked static or transient.

Another important issue is that RMI only supports synchronous method invocations. Asynchronous method invocations must be explicitly programmed, using threads.

Currently, the *PPC-VM* project is evaluating an efficient RMI for Java called KaRMI[5], which is part of the JavaParty project.

KaRMI features are described as follows:

> *"KaRMI is a fast drop-in replacement for the Java remote method invocation package (RMI). It is based on an efficient object serialization mechanism called uka.transport that replaces regular Java serialization from the java.io package. KaRMI and uka.transport are implemented completely in Java without native*

[5] http://www.ipd.uka.de/JavaParty/KaRMI/index.html

code. KaRMI also supports non-TCP/IP communication networks such as Myrinet/GM and Myrinet/ParaStation. It can also be used in clusters interconnected with heterogeneous communication technology."

2.2.3. Examples

Code 1 shows a simple example that creates several instances and calls their methods.

Code 1 Client class

```
public class Client {

    public static void main(String args[]) {
        for(int i=0; i<10; i++) {
            Server s = new Server();
            s.method1();  // calls method1
            s.method2(5); // calls method2 with argument 5
        }
    }
}
```

A class `Server` is defined in Code 2: both methods will print the host name where the instances were created.

Running `main` method from `Client` class in host `plutao.di.uminho.pt` we get the output illustrated in Output 1.

All instances were created in the same host, as Output 1 illustrates. Using the script presented in appendix B.2, we can distribute the instances among the nodes of a cluster. First we need to generate the support code and transform the `Server` class (using `pre` option). Then we need to generate the RMI stubs (using the `rmic` option). We use the `start` option to start the factories, the name server and the object manager. The nodes where factories are and some other information are referenced in a configuration file.

Output 2 shows the script in action.

We can run the `main` method from `Client` using the `flags` option. This option will inform the program where is the name server and the libraries it needs to run. Output 3 illustrates the client `main` method execution, using 5 nodes (`pe2`, `pe3`, `pe4`, `pe10`, `pe12`). The load distribution policy is round-robin and all the instances were distributed among the available nodes.

2.2.4. Limitations

The code generator still has some limitations, mostly due to some initial requirements. However, these limitations are being overcome, as presented below.

Code 2 Server class

```
public class Server {

    int number;

    public Server() {
        this.number = 0;
    }

    public Server(int a) {
        this.number = a;
    }

    public void method1() {
        try {
            java.net.InetAddress host =
                java.net.InetAddress.getLocalHost();
            String hostname = host.getHostName();
            System.out.println(
                "Method1 was called in host " + hostname);
        } catch (Exception e) {}
    }

    public void method2(int a) {
        try {
            java.net.InetAddress host =
                java.net.InetAddress.getLocalHost();
            String hostname = host.getHostName();
            System.out.println(
                "Method2 was called with argument: " +
                a +
                " in host " + hostname);
        } catch (Exception e) {}
    }
}
```

Output 1 Example in the same node

```
[joao@plutao demo]$ java Client
Method1 was called in host plutao.di.uminho.pt
Method2 was called with argument: 5 in host plutao.di.uminho.pt
Method1 was called in host plutao.di.uminho.pt
Method2 was called with argument: 5 in host plutao.di.uminho.pt
Method1 was called in host plutao.di.uminho.pt
Method2 was called with argument: 5 in host plutao.di.uminho.pt
Method1 was called in host plutao.di.uminho.pt
Method2 was called with argument: 5 in host plutao.di.uminho.pt
Method1 was called in host plutao.di.uminho.pt
Method2 was called with argument: 5 in host plutao.di.uminho.pt
Method1 was called in host plutao.di.uminho.pt
Method2 was called with argument: 5 in host plutao.di.uminho.pt
Method1 was called in host plutao.di.uminho.pt
Method2 was called with argument: 5 in host plutao.di.uminho.pt
Method1 was called in host plutao.di.uminho.pt
Method2 was called with argument: 5 in host plutao.di.uminho.pt
Method1 was called in host plutao.di.uminho.pt
Method2 was called with argument: 5 in host plutao.di.uminho.pt
Method1 was called in host plutao.di.uminho.pt
Method2 was called with argument: 5 in host plutao.di.uminho.pt
[joao@plutao demo]$
```

Output 2 Script used

```
[joao@plutao demo]$ ppcvm pre Server.java
Creating proxy file: Server.java
Creating implementation: ServerImpl.java
Creating interface: IServer.java
Creating factory: PPCFactory.java
Creating factory interface: IPPCFactory.java
                                                        [  OK  ]
[joao@plutao demo]$ ppcvm rmic
Running rmic in file: PPCFactory
Running rmic in file: ServerImpl                        [  OK  ]
                                                        [  OK  ]
[joao@plutao demo]$ ppcvm compile *.java
Compiling files:                                        [  OK  ]
[joao@plutao demo]$
```

Output 3 Example over multiple nodes

```
[joao@plutao demo]$ java `ppcvm flags` Client
Method1 was called in host pe2.gecinv.di.uminho.pt
Method2 was called with argument: 5 in host pe2.gecinv.di.uminho.pt
Method1 was called in host pe3.gecinv.di.uminho.pt
Method2 was called with argument: 5 in host pe3.gecinv.di.uminho.pt
Method1 was called in host pe4.gecinv.di.uminho.pt
Method2 was called with argument: 5 in host pe4.gecinv.di.uminho.pt
Method1 was called in host pe10.gecinv.di.uminho.pt
Method2 was called with argument: 5 in host pe10.gecinv.di.uminho.pt
Method1 was called in host pe12.gecinv.di.uminho.pt
Method2 was called with argument: 5 in host pe12.gecinv.di.uminho.pt
Method1 was called in host pe2.gecinv.di.uminho.pt
Method2 was called with argument: 5 in host pe2.gecinv.di.uminho.pt
Method1 was called in host pe3.gecinv.di.uminho.pt
Method2 was called with argument: 5 in host pe3.gecinv.di.uminho.pt
Method1 was called in host pe4.gecinv.di.uminho.pt
Method2 was called with argument: 5 in host pe4.gecinv.di.uminho.pt
Method1 was called in host pe10.gecinv.di.uminho.pt
Method2 was called with argument: 5 in host pe10.gecinv.di.uminho.pt
Method1 was called in host pe12.gecinv.di.uminho.pt
Method2 was called with argument: 5 in host pe12.gecinv.di.uminho.pt
```

Direct access to instance variables

Direct access to instance variables is not a good practice; however, some applications are written this way. The generated proxy does not have any instance variable; if some other class tries to directly access any instance variable, an error will occur.

The solution is to use accessors and mutators to read and to change instance variables; this way, the proxy will redirect the method calls to the implementation object.

Static variables

Current proxy generation not include any static or instance variables; if the client object tries to access a static variable, an error will occur.

Another problem arises if the server class has static variables and uses other classes, which try to access them: since the auxiliary classes are never changed, they will try to access the proxy's static variables. For instance, it is perfectly possible to have a class *Server* with a static variable named *var*, which uses an instance of class *Aux* that tries to access *Server.var*. Transforming class *Server* will produce the proxy *Server* (without static variables) and a new class *ServerImpl* (with the static variables). *Aux* class will not be changed and *Server.var* will result in an error.

The first thing we might think of is to change the call *Server.var* to *ServerImpl.var*. The problem is that if *Aux* object changes the *ServerImpl.var* value, this modification must be replicated to all the other *ServerImpl* objects that may be distributed among several nodes.

We have not found yet a final solution, but this problem will be solved in the future.

Inheritance

Class inheritance is not currently supported. If a class *Sub* extends another class *Super*, and if *Sub* does not redefine all *Super* methods, then the generated proxy will not direct the inherited and unredefined methods to the implementation object.

One way to overcome this problem is to aggregate all methods of all superclasses in the proxy; this requires to parse all superclasses in the pre-processing phase.

Note: if the used RMI form supported inheritance, the problem would be automatically overcome.

2.3. A C# implementation

There is also a C# implementation of the approach described in this chapter. It is called ParC# and it was developed on top of the Mono project[6]. The experiments with this implementation show that Mono C# Remoting presents a low latency, similar to highly optimized Java RMI implementations [NPH99]. However, the same experiments have shown that Mono's thread scheduling policy should be improved.

My contribution for this platform was to evaluate its performance; I tested it with a raytracing algorithm (ported from Java to C#) and compared the execution times with a Java RMI version. The full performance evaluation (and more details) is described in a scientific communication we presented in an international conference [FS05].

[6] http://www.mono-project.com

3. A Skeleton-based Java Framework

This chapter describes a skeleton-based Java framework built to help programmers to structure their parallel applications. It also introduces the skeletal approach to parallel programming by describing what a skeleton is and by presenting some common skeletons.

3.1. Parallel Skeletons

Many parallel algorithms share the same generic patterns of computation and interaction. Skeletal programming proposes that such patterns be abstracted and provided as a programmer's toolkit. We call these abstractions **algorithmical skeletons**, **parallel skeletons** or simply **skeletons**.

We define parallel skeletons as abstractions modelling a common, reusable parallelism exploitation pattern [ADT03]. A skeleton may also be seen as a high order construct (i.e. parameterized by other pieces of code and other parameters) which defines a particular parallel behaviour.

Skeletons provide an incomplete structure that can be parameterized by the number of processors, domain-specific code or data distribution; programmers can focus on the computational side of their algorithms rather than the control of the parallelism. Since the lower level operations are hidden, programmers' **productivity** increases.

Since skeletons provide simple interfaces to programmers, skeleton-based programs are smaller, easier to maintain, more understandable and less prone to error. These properties together with the fact that most parallel applications share the same interaction patterns, make skeletons a potential tool for code **reusability**.

Skeletons also provide a good way to code **portability**, because the same skeleton can be used for different architectures: it is only necessary to change the implementation of the skeleton in order to make a skeleton-based program work.

Usually, there is a trade-off between performance and reusability and portability. However, the skeletal approach provides programmers an easy way to optimise the computational part of their algorithms. Besides, skeletons may be carefully optimized to run more efficiently in the underlying architecture.

3.2. Common skeletons

Generally, skeletons can be divided in two main classes: **data parallel** skeletons and **task parallel** skeletons. Data parallel skeletons are based on a *distributed data structure*. Basically, the data is distributed among several processors and, usually, each processor executes

the same code on the different pieces of data. Task parallel skeletons are based on the distribution of the execution of independent tasks on several processors.

This section presents some common skeletons which capture the structure of most of the typical parallel applications.

3.2.1. Farm

The Farm skeleton is a data parallel skeleton and it consists of a master entity and multiple workers. The master decomposes the input data in smaller independent data pieces and send them to each worker. Workers process the data and send their result to the master, which merges them to get the final result.

Farm skeleton may use either a static load-distribution or a dynamic load-distribution. In the first case, all the data is distributed in the beginning of the computation. This strategy is suitable for homogeneous environments and for regular problems. The other approach is better to unbalanced problems or heterogeneous environments.

There is an interesting Farm skeleton variation, which uses a dynamic load-distribution, where data is sent only when workers demand it; this form is usually called *Dynamic Farm* or *Farm-on-Demand*. This variation is very useful for heterogeneous environments and when there is a large number of data pieces, since workers will be more efficiently used. However, communication costs are larger and performance may decrease.

A single master can be a bottleneck for a large number of workers, but skeletons can be tuned or changed to handle these limitations; a Farm skeleton can, for instance, use several masters to improve performance.

Figure 5 illustrates the Farm skeleton.

3.2.2. Pipeline

The Pipeline skeleton is a task parallel skeleton and it corresponds to the well known functional composition. The tasks of the algorithm are serially decomposed and each processor executes a task. Each processor/task is usually called a **stage**.

In most cases, input data are sent to the first stage and then flow between the adjacent stages of the pipeline. The computation ends when the last stage ends processing. However, the initial input data can also be decomposed in smaller blocks; then, each block is sent to the pipeline. This alternative uses more efficiently the workers, since the pipeline can process different data blocks at the same time.

Figure 6 illustrates the Pipeline skeleton.

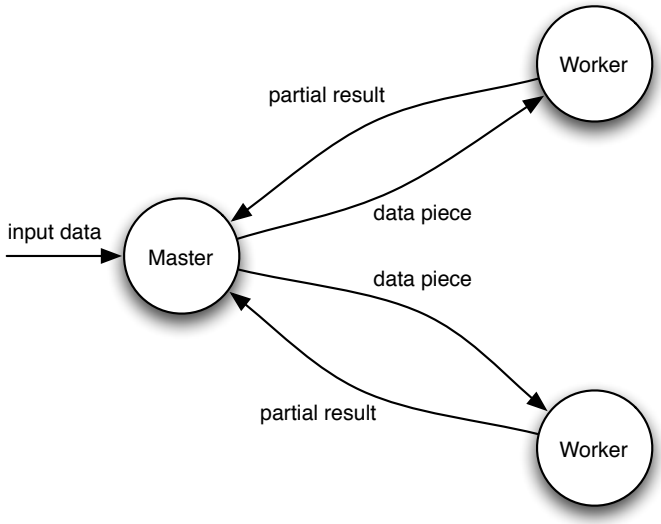


Figure 5: Farm Skeleton

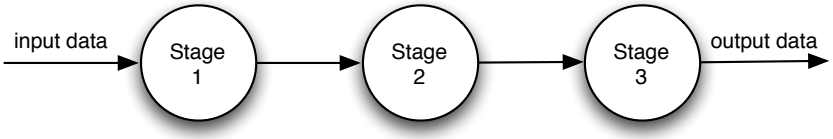


Figure 6: Pipeline Skeleton

3.2.3. Heartbeat

The Heartbeat skeleton models a very common pattern present in many parallel algorithms: data are spread among workers, each is responsible for updating a particular part and new data values depend on values held by other workers. It is called Heartbeat because the actions of each worker are like the beating of a heart: expand, sending information out; contract, gathering new information; then process the information and repeat [And99]. Heartbeat is appropriate for iterative algorithms and it is a communication-intensive skeleton.

Figure 7 illustrates the Heartbeat skeleton.

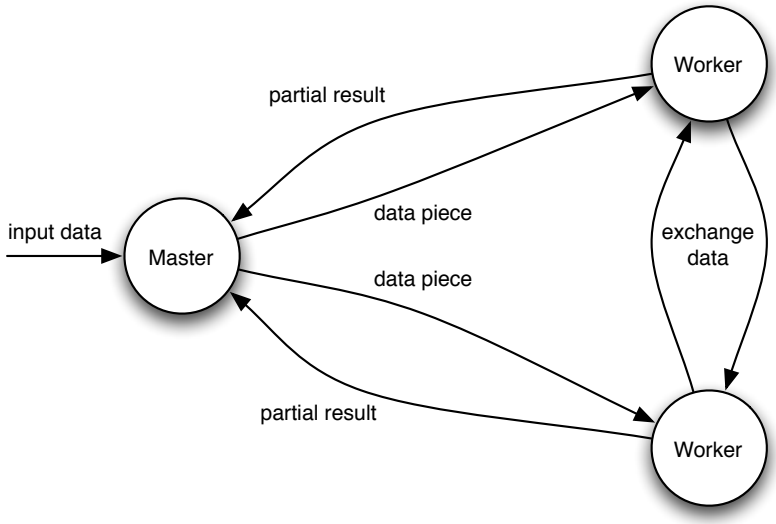


Figure 7: Heartbeat Skeleton

3.2.4. Divide-and-Conquer

The Divide-and-Conquer skeleton corresponds to the well known sequential algorithm with the same name. Basically, a problem is divided in subproblems and each of these subproblems is solved independently. Subproblems are independent from each other and they can be solved in different processors. The results of each subproblem are combined to get the final result.

Figure 8 illustrates the Divide-and-Conquer skeleton.

3.3. Skeletons composition

Conceptually, skeletons may be composed [DkGTY95, BC05] in order to get different interaction patterns. If a worker of a Farm can be expressed as a Heartbeat skeleton, then it seems a good idea to write it using the Heartbeat skeleton, because it will be more structured.

3.4. JaSkel: a skeleton-based Java framework

Skeletons can be provided to the programmer either as language constructs or as libraries. JaSkel provides parallel skeletons as a set of Java abstract classes.

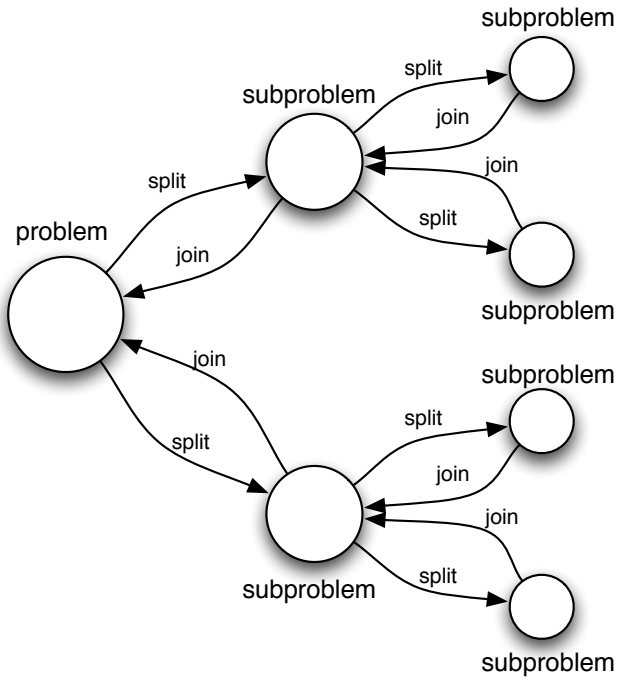


Figure 8: Divide-and-Conquer Skeleton

The only Java libraries for parallel programming based on skeletons that were found were *Lithium* [ADT03] and *muskel* [Dan05]. They are developed by the same research team and, according to *muskel* webpage[7], *Lithium* is no long being maintained:

> *"muskel is a full Java library allowing users to run task parallel computations onto networks/clusters of machines. It runs with Java 1.4 or higher. muskel is a core version of Lithium, which is no more maintained."*

The main differences between JaSkel and these two libraries are:

- JaSkel only provides a way to structure parallel applications; *muskel* and *Lithium* implement communication and distribution code;
- JaSkel explores class hierarchy and inheritance; *muskel* and *Lithium* are based on object composition.

[7] http://www.di.unipi.it/~marcod/muskel

The current JaSkel prototype provides skeletons for Farm and Pipeline parallel coding. Later versions will be extended to support other parallel skeletons or to improve current skeletons.

To write a parallel application using JaSkel, a programmer must perform the following steps:

- to structure the parallel program and to express it using the available skeletons;
- to refine the supplied abstract classes and write the domain-specific code used as skeleton parameters;
- to write the code that starts the skeleton, defining other relevant parameters (the number of processors, the load distribution policy, ...).

3.4.1. JaSkel API

The current JaSkel prototype provides the programmer different versions of the Farm and the Pipeline skeletons:

- a fully sequential Farm;
- a concurrent Farm that creates a new thread for each worker;
- a dynamic Farm, which sends only data to workers when they demand it;
- a fully sequential Pipeline;
- a concurrent Pipeline, which creates a new thread for each data flow.

A JaSkel skeleton is a simple Java class that implements the `Skeleton` interface and extends the `Compute` class. The interface `Skeleton` defines a method `eval` that must be defined by all the skeletons. This method starts the skeleton activity.

To create objects that will perform domain-specific computations, the programmer must create a subclass of class `Compute` (inspired in `muskel`). The `Compute` abstract class defines an abstract method `public abstract Object compute(Object input)` that defines the domain-specific computations involved in a skeleton.

For instance, to create a Farm, a programmer needs to perform the following steps:

- to create the worker's class, which is a subclass of `Compute`;
- to define the worker's inherited method `public Object compute(Object input);`
- to create the master's class which is a subclass of `Farm`;

- to define the methods `public Collection split(Object initialTask)` and `public Object join(Collection partialResults);`
- to create a new instance of the master's class and call the method `eval`; this method will basically perform the following steps:
  - it creates multiple workers;
  - it splits the initial data using the defined `split` method;
  - it calls `compute` method from each worker with the pieces of data returned by method `split`;
  - it merges the partial results using the defined `join` method.

Figure 9 shows the Farm skeleton UML class diagram. Some mutators and accessors were omitted.

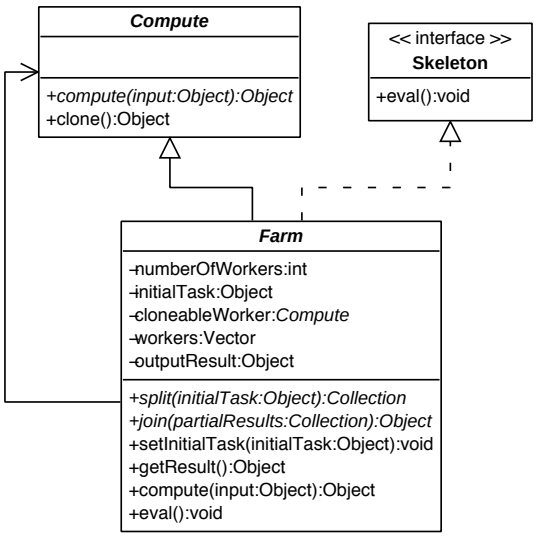


Figure 9: Sequential Farm skeleton: UML class diagram

The specialization or the creation of a new skeleton is done by class refinement. Figure 10 illustrates the parallel Farm skeleton UML class diagram, which extends the sequential Farm skeleton.

Either the skeletons or the entities that will perform domain-specific code extend the class `Compute`. Figure 11 illustrates the Pipeline skeleton, which also extends the `Compute` class.

JaSkel skeletons are also subclasses of `Compute` class to allow composition. Usually, the method `public Object compute(Object input)` on skeletons calls the `eval` method to start the skeleton activity.

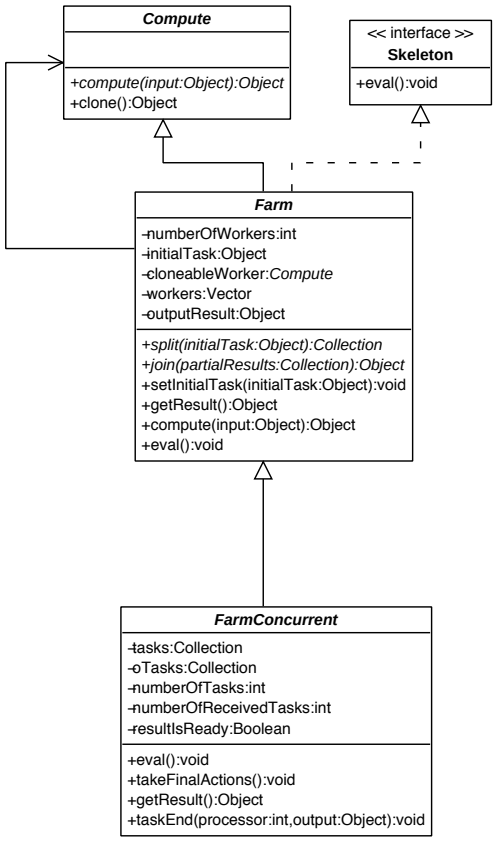


Figure 10: Parallel Farm skeleton: UML class diagram

3.4.2. Building skeleton-based applications

The best way to show how to build a skeleton-based application is through an example.

The problem: to find and count all prime numbers up to N.

A Solution[8]: begin with an (unmarked) array of integers from 2 to N. The first unmarked integer, 2, is the first prime. Mark every multiple of this prime. Repeatedly take the next unmarked integer as the next prime and mark every multiple of the prime. Note: Algorithm proposed by Eratosthenes of Cyrene (276 BC - 194 BC).

We have a Java implementation that implements this algorithm; it marks the multiples, setting them to 0. This implementation consists of two entities: a number generator and a prime filter. The first generates the input integer array $[2..N]$ and the latter filters the non-prime integers. A prime filter has a list with the primes from 2 to $sqrt(N)$ (called filter) and every

[8]Taken from http://www.nist.gov/dads/HTML/sieve.html

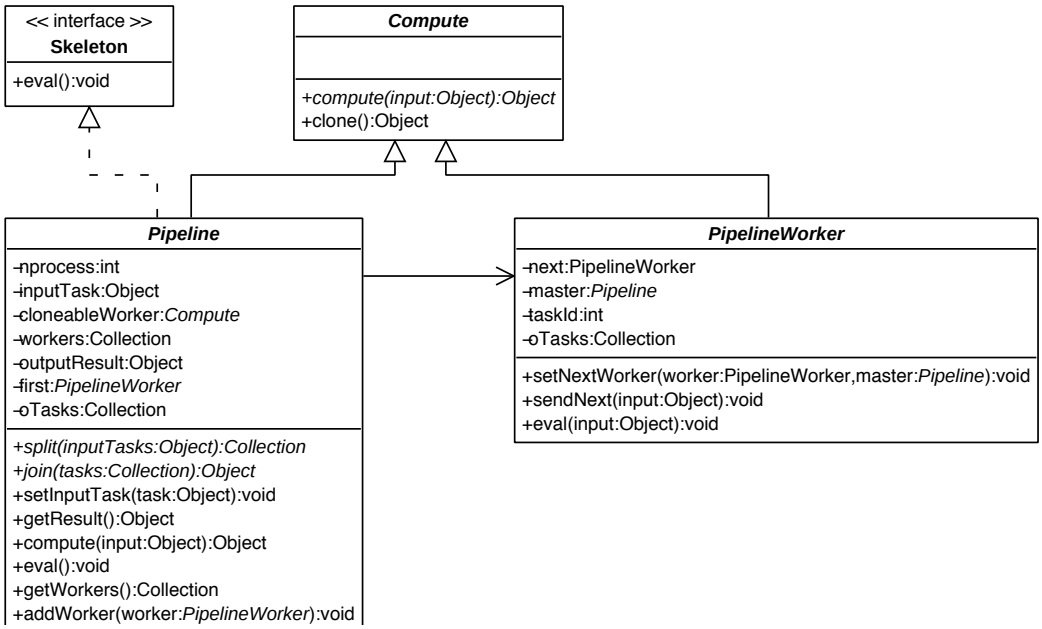


Figure 11: Pipeline skeleton: UML class diagram

integer n from the input array will be divided by each prime of this list; if it does not find any divisor, then n is prime.

This algorithm can be easily parallelized in two different ways:

- as a farm: the input array is decomposed in smaller pieces, and each piece is sent to a prime filter; each prime filter will test the integers received using the filter $[2..sqrt(N)]$;
- as a pipeline: each prime filter constitutes a pipeline stage and defines a different filter; the input data is sent to the first pipeline stage and then flows between the adjacent stages; when it reaches the end, all the non-primes integers were filtered.

The two next examples show how we can use the JaSkel framework to implement this algorithm as a Farm and as a Pipeline. The implementation will count the primes up to 10,000,000.

Primes sieve as a Farm

The prime filter (farm worker) is illustrated in Code 3. Its main method is `filter`, which filters the given integer array. The `compute` method, needed to define the skeleton's domain-

specific code, delegates its job to the method `filter`. Note that the class `PrimeFilter` is a subclass of `Compute`.

Code 4 illustrates the generator. It uses the skeleton `FarmConcurrent`, since it is its subclass. It defines the method `split` using the method `generate2` and it defines the method `join` as the identity method.

Code 5 shows the code that connects these entities. The performed steps are:

- it creates a prime filter object (`pf`) and initializes it (method `init`);
- it creates a new generator object (`g`), setting its parameters: the worker, the number of processes (`nprocess`) and input data;
- it starts the skeleton activity, calling method `eval`;
- it gets the final result, using method `getResult`.

Note that in this example, the farm receives a `null` input data because the `split` method already generates the integer blocks.

Output 4 illustrates the result of running the generator, with 4 workers.

Primes sieve as a Pipeline

The prime filter is illustrated in Code 6. The only difference between the farm and the pipeline prime filter is that the first is a subclass of `Compute`, and the latter is a subclass of `PipelineWorker`. The `PipelineWorker` class is a subclass of `Compute`, but it defines three new methods:

- `setNextWorker`, which sets the pipeline's stages;
- `sendNext`, which sends data to the next stage;
- `eval`, which calls method `compute` and makes the data flow between the adjacent stages.

The generator, illustrated in Code 7, is defined in the same way as the Farm generator, but it is a subclass of `PipelineConcurrent`.

Code 8 illustrates the code that connects these entities. The performed steps are:

- a list of prime filters is created (`workers`); all the filters are different and disjoint;

- it creates a new generator object (`g`), setting its parameters: the workers list (stages) and input data;
- it starts the skeleton activity, calling method `eval`;
- it gets the final result, using method `getResult`.

In this example, the pipeline also receives a `null` input data because the `split` method already generates the integer blocks.

Output 5 illustrates the result of running the generator, with 4 stages. Note that it creates four different prime filters.

3.4.3. Limitations

The current JaSkel prototype only provides one way to structure parallel applications. It will soon provide more complete and robust skeletons which automatically distribute workers among the nodes of a distributed environment.

Code 3 Prime filter: farm worker

```
package jaskel.examples.primes;
import java.util.*;
import jaskel.Compute;

public class PrimeFilter extends Compute {

        int[] myPrimes; // buffer to hold primes already calculated
        int nPrimes; // number of primes calculated
        PrimeFilter myNext; // next filter
        double start;
        int contaPrimos;
        int packs;
        int cupack = 0;
        int SMaxP;
        int myMaxP;
        int myMinP;

        public void init(int myMinP, int myMaxP, int SMaxP, PrimeFilter next,
                        int pac) {
                cupack = 0;
                myNext = next;
                nPrimes = 0;
                packs = pac;
                this.myMinP = myMinP;
                this.myMaxP = myMaxP;
                this.SMaxP = SMaxP;

                int[] pr = new int[SMaxP];
                int nl = PrimeCalc.lowPrimes(SMaxP, pr);

                myPrimes = new int[nl];
                for (int i = 0; i < nl; i++)
                        if (pr[i] >= myMinP && pr[i] <= myMaxP)
                                myPrimes[nPrimes++] = pr[i];
                System.out.println(nPrimes + " primes " + myMinP + "..." + myMaxP);
                contaPrimos = 0;
                start = new Date().getTime();
        }

        public synchronized int[] filter(final int[] num) {
                cupack++;
                for (int i = 0; i < num.length; i++) {
                        if (num[i] > 2) {
                                if (PrimeCalc.isPrime(num[i], myPrimes, nPrimes)) {
                                        contaPrimos++;
                                } else
                                        num[i] = 0;
                        }
                }
                return num;
        }

        /*
         * This method is different from implementation to implementation.
         *
         * @see jaskel.Compute#compute(java.lang.Object)
         */
        public Object compute(Object input) {
                return this.filter((int[]) input);
        }
}
```

Code 4 Generator: farm master

```
package jaskel.examples.primes;

import jaskel.Compute;
import jaskel.FarmConcurrent;
import java.util.Collection;
import java.util.Enumeration;
import java.util.Vector;

public class GeneratorFarm extends FarmConcurrent {

        int maxNumber;
        int sMAX;
        int blocksize;

        public GeneratorFarm(Compute worker, int nprocess, Object inputTask) {
                super(worker, nprocess, inputTask);
        }

        public Collection split(Object initialTask) {
                return this.generate2(sMAX + 1, maxNumber, blocksize);
        }

        public Object join(Collection partialResults) {
                return partialResults;
        }

        public Vector generate2(int iniNum, int maxNum, int blockSize) {
                int[] ar = new int[blockSize];
                int j = 0;
                Vector tasks = new Vector();

                for (int i = iniNum; i < maxNum; i += 2) {
                        ar[j++] = i;
                        if (j == blockSize) {
                                final int[] aux = ar;
                                tasks.add(aux);
                                j = 0;
                                ar = new int[blockSize];
                        }
                }
                for (int i = j; i < blockSize; i++)
                        ar[i] = 0;
                ar[ar.length - 1] = -3;
                tasks.add(ar);
                return tasks;
        }
}
```

Code 5 Generator: farm master's main method

```
public static void main(String[] args) {
    int nprocess = 4;
    int maxNumber = 10000000;
    int sMAX = (int) Math.sqrt(maxNumber);
    int blocksize = 100000;

    int MaxP = maxNumber;
    int SMaxP = sMAX;
    int packs = MaxP / (2 * blocksize);
    System.out.print("Primes up to " + MaxP + "; packages size " + blocksize / 2);
    System.out.print(" (" + MaxP / (2 * blocksize) + " packages - size ");
    System.out.println(4 * blocksize / 2 + " bytes)");

    PrimeFilter pf = new PrimeFilter();
    pf.init(1, SMaxP, SMaxP, null, packs);
    GeneratorFarm g = new GeneratorFarm(pf, nprocess, null);

    // Implementation details:
    g.blocksize = blocksize;
    g.maxNumber = maxNumber;
    g.sMAX = sMAX;

    // Starts the farming process and counts elapsed time
    long t0 = System.currentTimeMillis();
    g.eval();

    // Get the final result
    Object o = g.getResult();

    long t1 = System.currentTimeMillis();
    long elapsed = (t1 > t0 ? t1 - t0 : Long.MAX_VALUE - t0 + t1);
    System.out.println("Elapsed time " + elapsed + " millis");

    Enumeration e = ((Vector) o).elements();
    int soma = 0;
    while (e.hasMoreElements()) {
        int[] num = (int[]) e.nextElement();
        for (int i = 0; i < num.length; i++) {
            if (num[i] > 0)
                soma++;
        }
    }
    System.out.println("Number of primes: " + soma);
}
```

Output 4 Prime sieve as a Farm

```
Primes up to 10000000; packages size:  50000 (50 packages - size 200000 bytes)
445 primes 1...3162
Concurrent Farm Skeleton Active
Elapsed time 2770 millis
Number of primes: 664133
```

Code 6 Prime filter: pipeline stage

```
package jaskel.examples.primes;
import jaskel.PipelineWorker;
import java.util.Date;

public class PrimeFilter extends PipelineWorker {
        int[] myPrimes; // buffer to hold primes already calculated
        int nPrimes; // number of primes calculated
        PrimeFilter myNext; // next filter
        double start;
        int contaPrimos;
        int packs;
        int cupack = 0;
        int SMaxP;
        int myMaxP;
        int myMinP;

        public void init(int myMinP, int myMaxP, int SMaxP, PrimeFilter next, int pac) {
                cupack = 0;
                myNext = next;
                nPrimes = 0;
                packs = pac;
                this.myMinP = myMinP;
                this.myMaxP = myMaxP;
                this.SMaxP = SMaxP;

                int[] pr = new int[SMaxP];
                int nl = PrimeCalc.lowPrimes(SMaxP, pr);

                myPrimes = new int[nl];
                for (int i = 0; i < nl; i++)
                        if (pr[i] >= myMinP && pr[i] <= myMaxP)
                                myPrimes[nPrimes++] = pr[i];
                System.out.println(nPrimes + " primes " + myMinP + "..." + myMaxP);
                contaPrimos = 0;
                start = new Date().getTime();
        }

        public synchronized int[] filter(final int[] num) {
                cupack++;
                for (int i = 0; i < num.length; i++) {
                        if (num[i] > 2) {
                                if (PrimeCalc.isPrime(num[i], myPrimes, nPrimes)) {
                                        contaPrimos++;
                                } else
                                        num[i] = 0;
                        }
                }
                return num;
        }

        public Object compute(Object input) {
                return this.filter((int[]) input);
        }
}
```

Code 7 Generator: pipeline master

```
package jaskel.examples.primes;
import jaskel.PipelineConcurrent;
import java.util.Collection;
import java.util.Enumeration;
import java.util.Vector;

public class GeneratorPipeline extends PipelineConcurrent {
        int maxNumber;
        int sMAX;
        int blocksize;

        public GeneratorPipeline(Collection workers, Object inputTask) {
                super(workers, inputTask);
        }

        public Collection split(Object inputTask) {
                return this.generate2(sMAX + 1, maxNumber, blocksize);

        }

        public Object join(Collection tasks) {
                return tasks;
        }

        public Vector generate2(int iniNum, int maxNum, int blockSize) {
                int[] ar = new int[blockSize];
                int j = 0;
                Vector tasks = new Vector();

                for (int i = iniNum; i < maxNum; i += 2) {
                        ar[j++] = i;
                        if (j == blockSize) {
                                final int[] aux = ar;
                                tasks.add(aux);
                                j = 0;
                                ar = new int[blockSize];
                        }
                }
                for (int i = j; i < blockSize; i++)
                        ar[i] = 0;
                ar[ar.length - 1] = -3;
                tasks.add(ar);
                return tasks;

        }
}
```

Code 8 Generator: pipeline master's main method

```
public static void main(String[] args) {
    int nprocess = 4;
    int maxNumber = 10000000;
    int sMAX = (int) Math.sqrt(maxNumber);
    int blocksize = 100000;

    int MaxP = maxNumber;
    int SMaxP = sMAX;
    int packs = MaxP / (2 * blocksize);
    System.out.print("Primes up to " + MaxP + "; packages size " + blocksize / 2);
    System.out.print(" (" + MaxP / (2 * blocksize) + " packages - size ");
    System.out.println(4 * blocksize / 2 + " bytes)");

    PrimeFilter[] filtros = new PrimeFilter[nprocess];
    Vector workers = new Vector();
    try {
        for (int i = nprocess - 1; i >= 0; i--) {
            filtros[i] = new PrimeFilter();
            if (i != (nprocess - 1))
                filtros[i].init(i * SMaxP / nprocess + 1, (i + 1) * SMaxP
                                / nprocess, SMaxP, filtros[i + 1], packs);
            else
                filtros[i].init(i * SMaxP / nprocess + 1, (i + 1) * SMaxP
                / nprocess, SMaxP, null, packs);
         workers.add(filtros[i]);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }

    GeneratorPipeline g = new GeneratorPipeline(workers, null);

    // Implementation details:
    g.blocksize = blocksize;
    g.maxNumber = maxNumber;
    g.sMAX = sMAX;

    // Starts the farming process and counts elapsed time
    long t0 = System.currentTimeMillis();
    g.eval();

    // Get the final result
    Object o = g.getResult();

    long t1 = System.currentTimeMillis();
    long elapsed = (t1 > t0 ? t1 - t0 : Long.MAX_VALUE - t0 + t1);
    System.out.println("Elapsed time " + elapsed + " millis");

    Enumeration e = ((Vector) o).elements();
    int soma = 0;
    while (e.hasMoreElements()) {
        int[] num = (int[]) e.nextElement();
        for (int i = 0; i < num.length; i++) {
            if (num[i] > 0)
                soma++;
            }
        }
    System.out.println("Number of primes: " + soma);
}
```

Output 5 Prime sieve as a Pipeline

```
Primes up to 10000000; packages size: 50000 (50 packages - size 200000 bytes)
95 primes 2372...3162
102 primes 1582...2371
111 primes 791...1581
137 primes 1...790
New Concurrent Pipeline Skeleton Active
Elapsed time 11149 millis
Number of primes: 664133
```

4. Tests and Evaluation

Chapter 2 and Chapter 3 present an automatic object distribution platform and a skeleton-based Java framework. This chapter presents a comparative evaluation of alternative implementation strategies for the first component and an interesting comparison between two skeletons from the latter.

All the evaluation tests were run in a Linux cluster[9] with 16 nodes, connected through a Gigabit Ethernet. Each node is a bi-Xeon EM64T 3.2GHz with 2MB cache L2 and 2GB RAM. The Linux kernel version is the 2.6.9-5.0.5.ELsmp and the Java version is the 1.5.0_02.

4.1. Evaluation methodology

The Java Grande Forum Benchmark Suite[10] was selected to perform the comparative evaluation on the developed tools. The Java Grande Forum[11] (JGF) is a community initiative led by Sun and the Northeast Parallel Architectures Center (NPAC) that aims to promote the use of Java for so-called "Grande" applications. A Grande application is an application which has large requirements for memory, I/O, network bandwidth, or processing power. The benchmark suite provides ways of measuring and comparing alternative Java execution environments in ways which are relevant to Grande applications.

The benchmark suite consists of:

- sequential benchmarks, suitable for single processor execution;
- multi-threaded benchmarks, suitable for parallel execution on shared memory multiprocessors;
- MPI-based (MPJ) benchmarks, suitable for parallel execution on distributed memory multiprocessors;
- language comparison benchmarks, which are a subset of the sequential benchmarks translated into C.

Each of these benchmarks provide three benchmark types: low-level operations (referred as Section 1), simple kernels (Section 2) and applications (Section 3). The low-level operations benchmarks test the performance of low-level operations that will ultimately determine the performance of real Java applications. The simple kernels are small applications that are

[9] SeARCH cluster, hosted at Departamento de Informática - Universidade do Minho
[10] http://www.epcc.ed.ac.uk/javagrande/javag.html
[11] http://www.javagrande.org

commonly used in Grande applications, such as FFTs, LU Factorisation, sorting and searching. The application benchmarks represent Grande applications, such as a raytracer and a financial simulation, using Monte Carlo techniques [BSW+00].

The validation of the *PPC-VM* project tools is based on the JGF Benchmark suite. This work contributes for that validation in two different ways: by providing detailed documentation about the MPJ benchmarks (appendix A) and by testing the tools described in this report using some of the benchmark suite examples.

The evaluation of the automatic object distribution platform follows the same approach as the JGF Benchmark suite: it is based on low-level and on high-level tests.

The low-level evaluation measures the base communication latency and bandwidth. It is based on a ping-pong test, which measures the costs of point-to-point communication for a range of message lengths. This test is equivalent to the PingPong test provided by the JGF MPJ benchmarks (Section 1). Bandwidth measures the rate at which data is passed over the network and latency measures the amount of time a message takes to get from the source node to the destination node.

The high-level evaluation measures the performance of a raytracing parallel algorithm, adopted from the MPJ benchmark suite (Section 3).

4.2. Automatic object distribution platform

All performed tests compare the RMI-IIOP based platform with RMI-IIOP versions developed from scratch. We also show the values for the platform's current version, based on KaRMI. Each test was executed 11 times, and the presented values correspond to the median value.

4.2.1. Low-level evaluation

Tables 1 and 2 show the obtained latency and bandwidth values. Figure 12 illustrates the relation between these values.

The RMI-IIOP based platform performance is always worst than the RMI-IIOP version: in some cases the performance degradation is superior to 100%, but for the largest message, the performance is only 11% worst. The performance degradation was expected, since the platform has the additional steps of deciding where to create the remote objects and then proceed to their creation.

The KaRMI based platform is the one which achieves the best performance, since it is the most optimized RMI form. KaRMI is based on an efficient object serialization mechanism called `uka.transport` that replaces regular Java serialization from the `java.io` package [NPH99].

Size (bytes)	Time (microseconds)		
	RMI-IIOP	**RMI-IIOP Platform**	**KaRMI Platform**
0	1730	3787	250
8	1585	2175	250
100	1625	2387	250
1000	1935	5500	562
10000	4880	9162	1087
100000	17500	51537	2862
1000000	310500	346225	58662

Table 1: Latency values

Size (bytes)	Bandwidth (MB/s)		
	RMI-IIOP	**RMI-IIOP Platform**	**KaRMI Platform**
0	0.0005	0.0005	0.0040
8	0.0062	0.0046	0.0400
100	0.0601	0.0419	0.4000
1000	0.5047	0.1818	1.7777
10000	2.0011	1.0914	9.1954
100000	5.5803	1.9403	34.9345
1000000	3.1451	2.8882	17.0467

Table 2: Bandwidth values

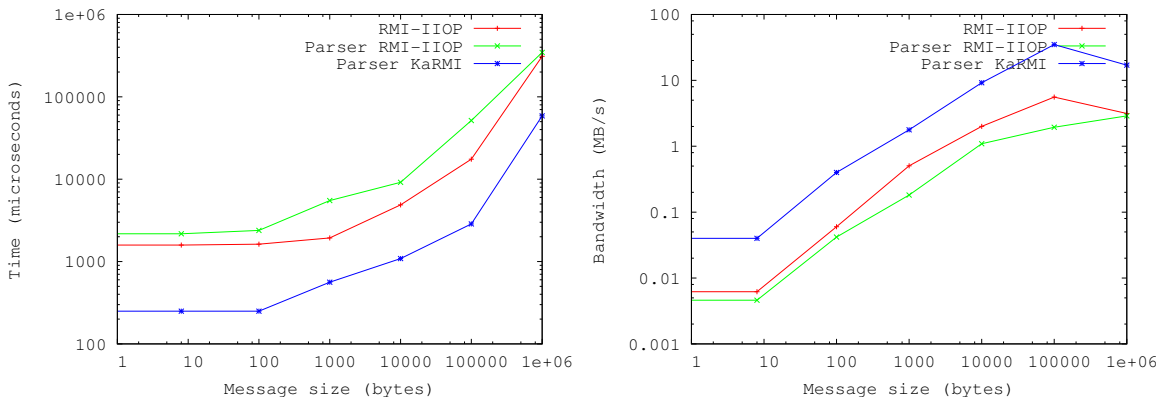
Note: the obtained results are worst than expected, probably because the SeARCH cluster is still in under tests.

4.2.2. High-level evaluation

Java Grande Forum Raytracer

The Java Grande Forum benchmark suite provides a raytracer parallel algorithm (described in appendix A), which renders a scene with 64 spheres. We have created two new versions: a RMI-IIOP equivalent implementation and a sequential version capable of distributing the work among instances. The latter was used to test the automatic object distribution platform.

Tables 3, 4 and 5 show the execution times of the Java Grande Forum raytracer algorithm to render scenes with 500x500, 1000x1000 and 2000x2000 pixels. The compared versions are the RMI-IIOP version implemented from scratch and the RMI-IIOP and KaRMI based platforms.



(a) Latency (b) Bandwidth

Figure 12: Low-level tests

The raytracer results show that for larger images, the differences between the versions are smaller. Rendering a 1000x1000 pixels image (Table 4), the RMI-IIOP version is 15% or 20% better than the RMI-IIOP based platform. The KaRMI based platform is about 10% more efficient than the RMI-IIOP version and almost 20% more efficient than the RMI-IIOP based platform.

Rendering a 2000x2000 pixels image (Table 5), the RMI-IIOP version is 10% or 15% better than the RMI-IIOP based platform. The KaRMI based platform presented very similar results to the RMI-IIOP version; in fact, for 32 processors, the RMI-IIOP version achieved the best performance (8% better than the KaRMI platform). The KaRMI based platform performance is about 5% better than the RMI-IIOP based platform one.

For the 500x500 pixels image rendering (Table 3), the differences between the several versions are more noticeable. The main reason is that the communication time takes a significant amount of the total time.

Number of processors	Time (seconds)		
	RMI-IIOP	**RMI-IIOP Platform**	**KaRMI Platform**
1	51.834	54.387	44.419
2	32.075	30.394	25.831
4	18.663	18.077	17.368
8	14.584	15.610	10.906
16	10.631	13.807	10.510
32	10.512	16.269	8.480

Table 3: JGF Raytracer execution times: 500x500 image

Number of processors	Time (seconds)		
	RMI-IIOP	**RMI-IIOP Platform**	**KaRMI Platform**
1	197.370	218.645	208.429
2	120.003	114.437	114.784
4	62.047	66.189	61.068
8	34.853	39.534	33.204
16	24.186	25.596	21.992
32	18.410	23.513	16.760

Table 4: JGF Raytracer execution times: 1000x1000 image

Number of processors	Time (seconds)		
	RMI-IIOP	**RMI-IIOP Platform**	**KaRMI Platform**
1	937.712	842.490	827.900
2	450.958	424.776	433.739
4	227.936	236.235	230.199
8	124.985	120.942	121.742
16	70.054	72.733	65.969
32	41.661	47.551	45.066

Table 5: JGF Raytracer execution times: 2000x2000 image

Figures 13 and 14 illustrate the speedup curves for the raytracer results. The formula to calculate the speedup is very simple:

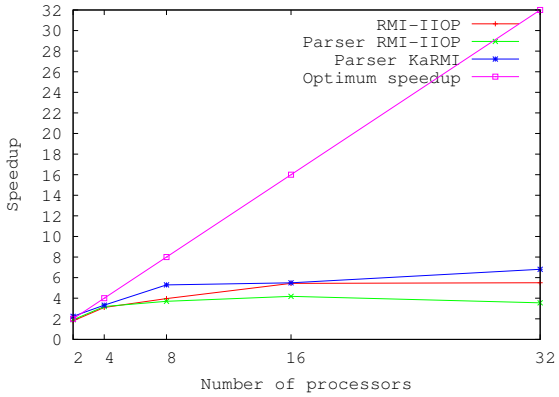
$$Speedup = \frac{Sequential\ time}{Parallel\ time}$$

The sequential time is the execution time of the sequential version.

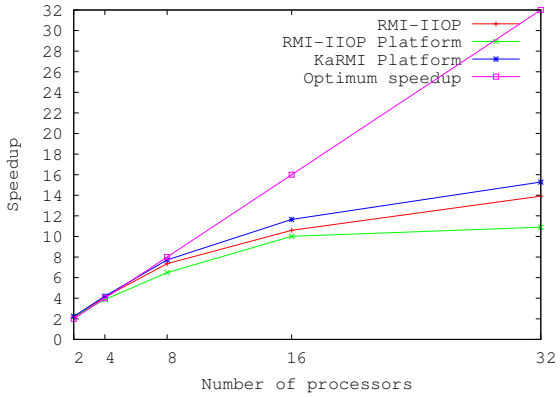
Figure 13 shows that the larger the image is, better is the speedup. Since the raytracer is a farming, we observe efficiency degradation for smaller images because of relatively increased communication overhead.

For the 2000x2000 image rendering, the RMI-IIOP based platform speedup is almost linear up to 16 processors; using 32 processors, the efficiency is of 60%. Rendering a 500x500 pixels image, the RMI-IIOP based platform speedup is very poor, specially when compared with the KaRMI based platform speedup.

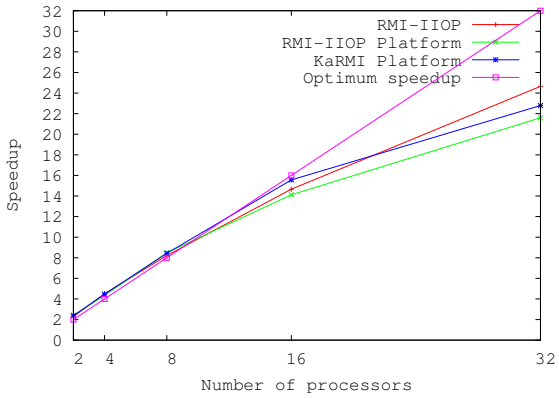
The KaRMI based platform results are better, since it is the most efficient RMI form. However, and strangely, for the 2000x2000 image rendering with 32 processors, the RMI-IIOP version presented the best execution time.



(a) 500x500 Image

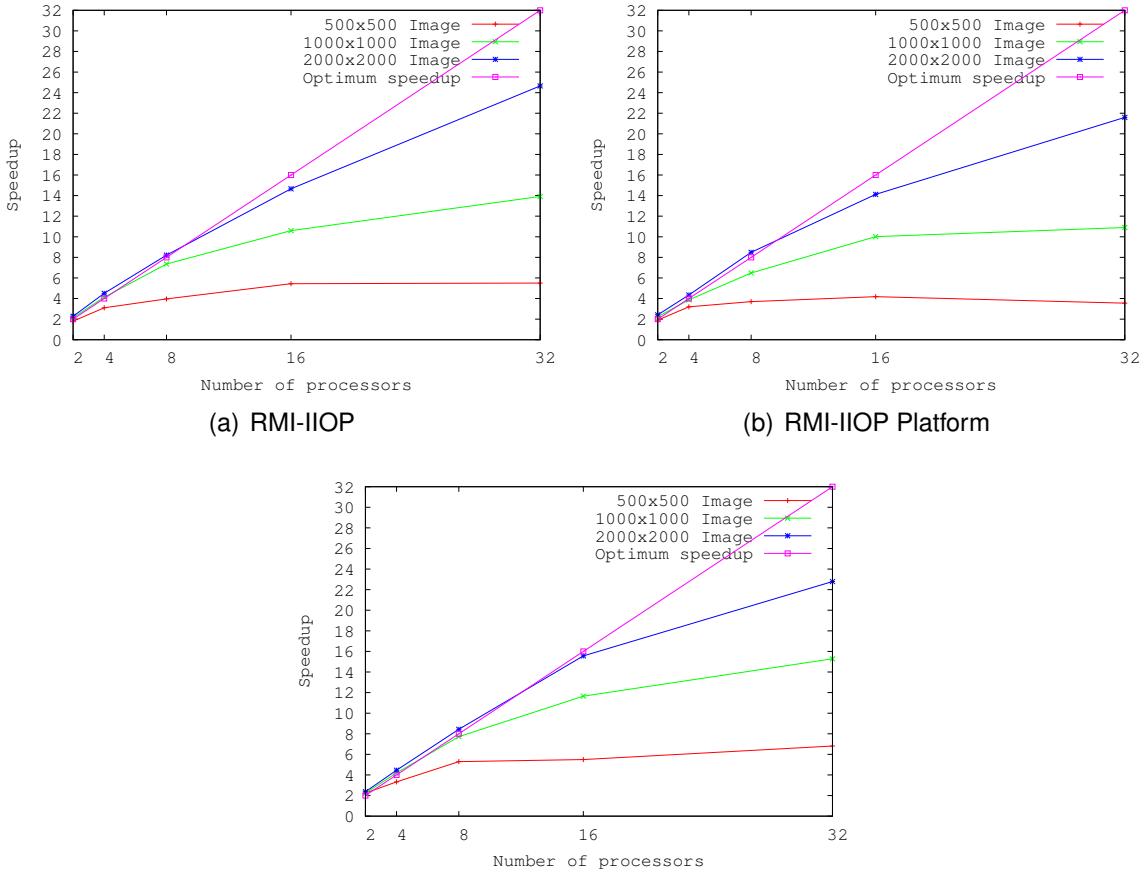


(b) 1000x1000 Image



(c) 2000x2000 Image

Figure 13: JGF RayTracer speedup (by image size)



(a) RMI-IIOP

(b) RMI-IIOP Platform

(c) KaRMI Platform

Figure 14: JGF RayTracer speedup (by type of implementation)

4.3. JaSkel evaluation

It is not currently possible to automatically distribute code that is structured using the JaSkel framework. However, we have some results that show an interesting comparison between the same algorithm implemented as a Farm and as a Pipeline, using one bi-processor node.

Primes sieve comparison

Figure 15 illustrates the execution times of the primes sieve algorithm implemented as a Farm and as a Pipeline (as shown in section 3.4.2).

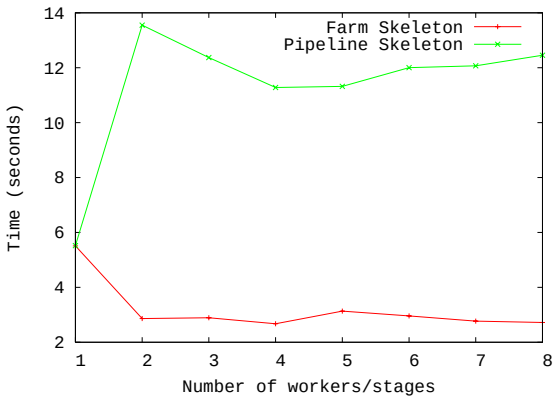


Figure 15: Primes sieve execution times, up to 10,000,000

The Farm and Pipeline execution times for one worker/stage are identical, since a Pipeline with one stage is equivalent to a Farm with one worker. However, for a larger number of workers/stages, the results are unexpectedly very different: the Pipeline execution times are more than four times longer.

The Pipeline skeleton requires much more communication: for n stages, the Pipeline skeleton requires $(1 + \mathit{nstages}) \times \mathit{packages}$ messages, since each package passes by all the stages and at the end it is returned to the master; the Farm skeleton requires $2 \times \mathit{packages}$ messages, since each package is sent to a worker and then it is returned to the master. However, these tests were executed in a shared memory environment where there is no data movement.

From these results it seems that the prime sieve algorithm is not suitable to be implemented following a Pipeline approach. However, we have a manually tuned version implemented as a Pipeline that has similar execution times to those presented by the Farm alternative; this suggests that the Pipeline skeleton may have a flaw that requires further analysis and code rewriting, to be performed as soon as the cluster nodes are stable enough to guarantee evaluation results.

5. Conclusions and Future Work

The main goal of this project was to contribute to a distributed Java virtual machine, providing an integrated approach to parallel computing, where the skeleton-based framework is used to structure parallel applications and the automatic object distribution platform is used to distribute objects efficiently. However, the connection between these two components is not yet implemented. This is the main drawback of the presented work.

The automatic object distribution platform that was developed satisfies the initial requirements: it distributes objects automatically among the nodes of a distributed environment and it is extensible. The first requirement is successfully proved in this report. The second requirement was also satisfied, since the platform was extended by other members of the *PPC-VM* project to improve the base communication, to gather information about the distributed environment and to make better decisions when distributing objects. Some results from the current version based on KaRMI were also shown.

The tests have shown that the developed platform may be less efficient than implementations manually tuned, but the development time reduction outweighs the performance degradation - at least, for the studied examples.

The developed skeleton-based framework presents a way to structure parallel OO applications worth pursuing; current prototype still provides few skeletons and the results show that the Pipeline skeleton needs improvements.

This work also contributes with documentation about the Java Grande Forum parallel algorithms. We do not know any document that describes all the JGF parallel algorithms implementation; we believe that this is a valuable document for someone who needs to understand any of these implementations.

5.1. Future work

The work described in this report is far from complete and should be continued and extended. As future work, we think that the most important things that must be done are:

- **to connect JaSkel structured code with the automatic object distribution platform:** this is one of the most important improvements that must be done; we believe that the JaSkel framework is a valuable component that will help programmers to maintain and to optimize their applications, but if we do not provide the connection between the framework and the automatic object distribution platform, programmers will not benefit from it;
- **to implement all Java Grande Forum parallel algorithms using JaSkel:** this will validate even more the work and may arise some new interesting issues not yet con-

sidered;

- **to test and optimize the automatic object distribution platform:** there is already a KaRMI based version that improves the platform's performance, but the information gathering about the distributed environment must be improved;
- **to implement new skeletons in the JaSkel framework:** the skeleton-based framework needs to implement a larger number of skeletons, so that a larger number of applications can be implemented using the framework; two of the more important skeletons that must be implemented are the Heartbeat and the Divide-and-Conquer skeletons.

Acronyms

ANTLR ANother Tool for Language Recognition

API Application Program Interface

AST Abstract Syntax Tree

CORBA Common Object Request Broker Architecture

DJVM Distributed Java Virtual Machine

IIOP Inter-ORB Protocol

JGF Java Grande Forum

JIT (compilation) Just-in-time (compilation)

MPI Message Passing Interface

ORB Object Request Broker

RMI Remote Method Invocation

RMI-IIOP Remote Method Invocation over IIOP

UML Unified Modeling Language

VM Virtual Machine

References

[ADT03] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Gener. Comput. Syst.*, 19(5):611–626, 2003.

[And99] Greg R Andrews. *Foundations of Parallel and Distributed Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[BC05] A. Benoit and M. Cole. Two fundamental concepts in skeletal parallel programming. In V. Sunderam, D. van Albada, P. Sloot, and J. Dongarra, editors, *The International Conference on Computational Science (ICCS 2005), Part II*, LNCS 3515, pages 764–771. Springer Verlag, 2005.

[BSW+99] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A methodology for benchmarking Java Grande applications. In *Java Grande*, pages 81–88, 1999.

[BSW+00] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.

[Col04] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, 2004.

[Dan05] Marco Danelutto. Qos in parallel programming through application managers. In *PDP*, pages 282–289, 2005.

[DkGTY95] John Darlington, Yi ke Guo, Hing Wing To, and Jin Yang. Parallel skeletons for structured composition. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 19–28, New York, NY, USA, 1995. ACM Press.

[FS05] João Fernando Ferreira and João Luís Sobral. ParC#: Parallel computing with C# in .Net. *Lecture Notes in Computer Science*, 3606:239–248, 2005.

[NPH99] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 152–159, New York, NY, USA, 1999. ACM Press.

[SBO01] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel Java Grande benchmark suite. pages ??–??, 2001.

A. Java Grande Forum MPJ Benchmarks

This document describes a subset of algorithms from the Java grande Forum MPJ Benchmarks. The descriptions emphasize the implementations parallelism, describing where and which data is exchanged between processors. The benchmark suite design description is out of the scope of this document. For more informations about this subject, see [BSW+99, BSW+00, SBO01]

A.1. Section 2: Kernels

A.1.1. Series

Algorithm description

Periodic functions may be represented in terms of an infinite sum of sines and cosines.The computation and study of Fourier series is known as harmonic analysis and is extremely useful as a way to break up an arbitrary periodic function into a set of simple terms that can be plugged in, solved individually, and then recombined to obtain the solution to the original problem or an approximation to it to whatever accuracy is desired or practical.

According to the theory developed by Fourier, any periodic function $F(t)$, with period T, may be represented by an infinite series of the form.

$$F(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \, cos \, n\omega_T t + b_n \, sin \, \omega_T t$$

where the coefficients a_0,a_n and b_n for a given periodic function $F(t)$ are calculated by the formulas

$$\omega_T = \frac{2\pi}{T}$$

$$a_0 = \frac{2}{T} \int_0^T F(t) \, dt$$

$$a_n = \frac{2}{T} \int_0^T F(t) \, cos \, n\omega_T t \, dt \qquad n = 1, 2, \ldots$$

$$b_n = \frac{2}{T} \int_0^T F(t) \, sin \, n\omega_T t \, dt \qquad n = 1, 2, \ldots$$

This series is called the Fourier series and the coefficients are called the Fourier coefficients.

The JGF benchmark algorithm computes the first N Fourier coefficients of the function $F(x) = (x+1)^x$ on the interval $0, 2$, where N is an arbitrary number that is set to make the test last long enough to be accurately measured by the system clock. Results are reported in number of coefficients calculated per second.

Algorithm methods

All the algorithms in JGF benchmark suite include control classes that initialise input data, implement validation methods and in some cases they even do data distribution among processes.

This algorithm's main class is `SeriesTest` and its control class is `JGFSeriesBench`. The control class determines the array size (`p_array_rows`) on each process (method `JGFinitialize`), as Code 9 illustrates.

Code 9 Series algorithm: partial arrays initialization

```
42  public void JGFinitialise(){
43    array_rows = datasizes[size];

45    /* determine the array dimension size on each process
46       p_array_rows will be smaller on process (nprocess-1).
47       ref_p_array_rows is the size on all processes except process (nprocess-1),
48       rem_p_array_rows is the size on process (nprocess-1).
49    */

51    p_array_rows = (array_rows + nprocess -1) / nprocess;
52    ref_p_array_rows = p_array_rows;
53    rem_p_array_rows = p_array_rows - ((p_array_rows*nprocess) - array_rows);
54    if(rank==(nprocess-1)){
55      if((p_array_rows*(rank+1)) > array_rows) {
56        p_array_rows = rem_p_array_rows;
57      }
58    }

60    buildTestData();
61  }
```

`buildTestData` method creates `TestArray` array on process rank 0 and creates `p_TestArray` array on every process (the size of this array is determined in the data initialisation in class `JGFSeriesBench`). Code 10 shows how `buildTestData` method is implemented and Figure 16 illustrates the data distribution.

After data initialisation, `Do` method is called (defined in line 85). This is the main method that will calculate the first n pairs of Fourier coefficients of the function $(x+1)^x$ on the interval $0, 2$. n is given by variable `array_rows` and the number of integration steps is fixed to 1000.

Code 10 Series algorithm: arrays creation

```
64  void buildTestData()
65  {
66      // Allocate appropriate length for the double array of doubles.
67
68      if(JGFSeriesBench.rank==0) {
69        TestArray = new double [2][array_rows];
70      }
71      p_TestArray = new double [2][p_array_rows];
72  }
```

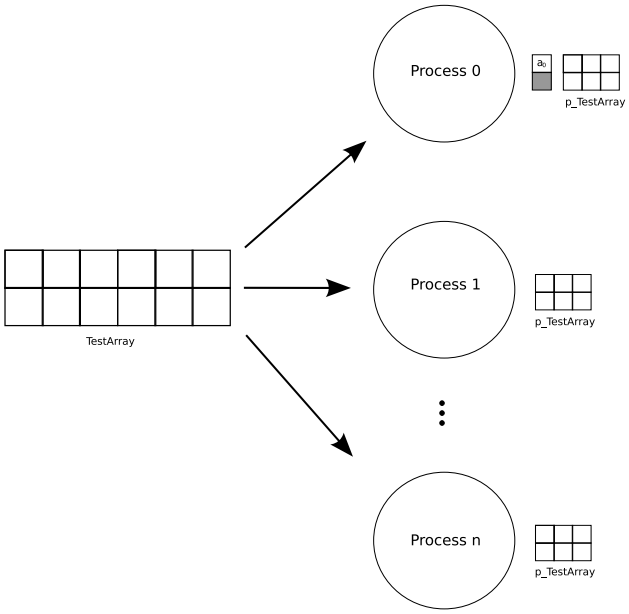


Figure 16: Data distribution in Series algorithm

Algorithm parallelization

The most time consuming component of the benchmark is the loop over the Fourier coefficients. Each iteration of the loop is independent of every loop and the work may be distributed simply between processes. Parallelism is in `Do` method and `TestArray` variable is the result array containing all the coefficients pairs (a_n, b_n).

Process rank 0 calculates a_0 (line 96) and it is the responsible for joining all the partial

results. All the processes calculate different coefficients independently and put them in a partial result array named `p_TestArray`. This array is sent to process rank 0 to be merged in `TestArray`, as Code 11 illustrates.

Code 11 Series algorithm: data merging

```
137  MPI.COMM_WORLD.Barrier();
138
139  // Send all the data to process 0
140
141  if(JGFSeriesBench.rank==0) {
142
143    for(int k=1;k<p_array_rows;k++){
144      TestArray[0][k] = p_TestArray[0][k];
145      TestArray[1][k] = p_TestArray[1][k];
146    }
147
148    for(int k=1;k<JGFSeriesBench.nprocess;k++) {
159
150      MPI.COMM_WORLD.Recv(p_TestArray[0],0,p_TestArray[0].length,MPI.DOUBLE,k,k);
151      MPI.COMM_WORLD.Recv(p_TestArray[1],0,p_TestArray[1].length,MPI.DOUBLE,k,
152      k+JGFSeriesBench.nprocess);
153
154      if(k==(JGFSeriesBench.nprocess-1)) {
155        p_array_rows = rem_p_array_rows;
156      }
157
158      for(int j=0;j<p_array_rows;j++){
159        TestArray[0][j+(ref_p_array_rows*k)] = p_TestArray[0][j];
160        TestArray[1][j+(ref_p_array_rows*k)] = p_TestArray[1][j];
161      }
162
163    }
164
165    p_array_rows = ref_p_array_rows;
166
167  } else {
168
169    MPI.COMM_WORLD.Ssend(p_TestArray[0],0,p_TestArray[0].length,MPI.DOUBLE,0,
       JGFSeriesBench.rank);
170    MPI.COMM_WORLD.Ssend(p_TestArray[1],0,p_TestArray[1].length,MPI.DOUBLE,0,
171      JGFSeriesBench.rank+JGFSeriesBench.nprocess);
172  }
```

The process interaction model used by this algorithm is usually called **Farming** because there is a manager process which splits initial data in a set of independent tasks, distributes each task by a different worker and in the end it collects the results.

A.1.2. LU factorisation

Algorithm description

Gaussian elimination transforms the system $Ax = b$ into an equivalent system $Ux = y$, where U is an upper triangular matrix. To perform this transformation, we calculate a sequence of multipliers.

If we want to calculate the system $Ax = c$ using simple gaussian elimination, then we need to perform the method from the beginning, calculating all the multipliers again.

LU factorisation solves this problem. Basically, it stores all the multipliers in a lower triangular matrix L that has ones on the main diagonal and zeros above the diagonal and where every other element $L[j,i]$ is the multiplier $A[j,i]/pivot$ (where $pivot$ is the value of the pivot element used for column i). In the end, the matrix product of L and U will be equal to the matrix A.

Having L and U matrices calculated, we have:

$$\begin{aligned} Ax &= b \quad \Leftrightarrow \\ (LU)x &= b \quad \Leftrightarrow \\ L(Ux) &= b \end{aligned}$$

So, calculating the system $Ax = b$ is the same as calculating the following two systems:

$$Ly = b \tag{1}$$
$$Ux = y \tag{2}$$

We can use forward substitution to solve 1 for y and then use back substitution to solve 2 for x.

The great advantage of this method is that once we have computed L and U, we can easily solve the system $Ax = b$ for different right-hand sides b.

The JGF benchmark algorithm solves a $N \times N$ linear system using LU factorisation followed by a triangular solve. It is a Java version of the well known Linpack benchmark. Performance units are Mflops per second.

Algorithm methods

This algorithm's main class is `Linpack` and its control class is `JGFLUFactBench`. In the control class, method `JGFinitialise` is the responsible for the data initialisation. It creates matrices `a`, `b` and `x` in process rank 0 and it determines the size of the sub arrays and copy the data in a cyclic manner to the sub array `buf_a`.

The other two main methods are `dgefa` and `dgesl`. `dgefa` method factors a double precision matrix by gaussian elimination and it is done in parallel. `dgesl` method solves the double precision system $a * x = b$ or $trans(a) * x = b$ using the factors computed by dgefa. This method is executed in the process rank 0.

This algorithm uses Gaussian elimination with *partial pivoting*, which is used to minimize the growth of machine roundoff error during a solution. Partial pivoting is the interchanging of rows in order to place a particularly "good" element in the diagonal position prior to a particular operation. The implementation takes notes of each switch in equations (that's the reason for extra integer array `ipvt`).

Algorithm parallelization

LU factorisation is the only part of the algorithm that is done in parallel. The forward and back substitutions to solve the system are done in process rank 0 in serial. Thus, `dgesl` method is executed only in process rank 0.

The initial data distribution is made in the method `JGFinititialise` where process rank 0 sends the matrix lines in a cyclic fashion to every process. Each process will store its lines in array `buf_a`. Code 12 shows how the data is distributed and Figure 17 illustrates it.

Code 12 LU Factorisation algorithm: data distribution

```
87   if(rank==0) {
88     r_count = 0;
89     z_count = 0;
90     for(int i=0;i<a.length;i++){
91       if(r_count==0) {
92         for(int l=0;l<a[0].length;l++){
93           buf_a[z_count][l] = a[i][l];
94         }
95         z_count++;
96       } else {
97         MPI.COMM_WORLD.Send(a,i,1,MPI.OBJECT,r_count,10);
98       }
99
100      buf_list[i] = z_count - 1;
101      list[i] = r_count;
102      r_count++;
103      if(r_count == nprocess) {
104        r_count = 0;
105      }
106
107      }
108
109  } else {
110    for(int i=0;i<real_p_ldaa;i++){
111      MPI.COMM_WORLD.Recv(buf_a,i,1,MPI.OBJECT,0,10);
112    }
113    for(int i=real_p_ldaa;i<buf_a.length;i++){
114      for(int ki=0;ki<buf_a[0].length;ki++){
115        buf_a[i][ki] = -9.0;
116      }
117    }
118
119  }
120
121  MPI.COMM_WORLD.Bcast(list,0,list.length,MPI.INT,0);
122  MPI.COMM_WORLD.Bcast(buf_list,0,list.length,MPI.INT,0);
```

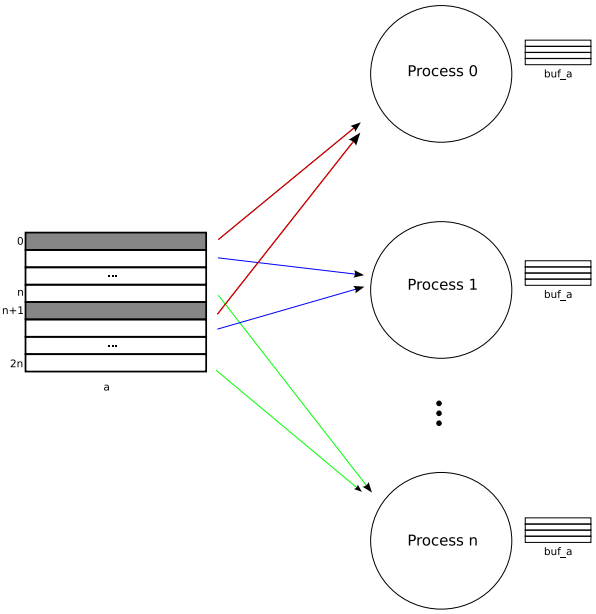


Figure 17: Data distribution in LU factorisation algorithm

This method also broadcasts arrays `list` and `buf_list` (lines 121 and 122). `list[i] = p` means that process rank p has received the i^{th} line.

In method `dgefa`, before the row elimination with column indexing, there are two arrays that are broadcasted: an array of doubles (`buf_col_k`) and an array of integers (`tmp_l`). Code 13 illustrates this step.

Code 13 LU Factorisation algorithm: intermediate arrays broadcasting

```
219  /* Broadcast the copy buf_col_k to all processes */
220
221  MPI.COMM_WORLD.Bcast(buf_col_k,0,buf_col_k.length,MPI.DOUBLE,list[k]);
222  tmp_l[0] = l;
223  MPI.COMM_WORLD.Bcast(tmp_l,0,tmp_l.length,MPI.INT,list[k]);
224  l = tmp_l[0];
```

At the end of the method `dgesl`, each process send to process rank 0 its partial results. Process rank 0 receives those results and collects them in array `a`, as Code 14 illustrates.

Code 14 LU Factorisation algorithm: data merging

```
254  if(JGFLUFactBench.rank==0) {
255    z_count = 0;
256    for(int i=0;i<a.length;i++){
257      if(list[i]==JGFLUFactBench.rank) {
258        for(int jl=0;jl<a[0].length;jl++){
259          a[i][jl] = buf_a[z_count][jl];
260        }
261        z_count++;
262      } else {
263        MPI.COMM_WORLD.Recv(a,i,1,MPI.OBJECT,list[i],10);
264      }
265    }
266  } else {
267    for(int i=0;i<JGFLUFactBench.real_p_ldaa;i++){
268      MPI.COMM_WORLD.Send(buf_a,i,1,MPI.OBJECT,0,10);
269    }
270  }
```

A.1.3. SOR: successive over-relaxation

Algorithm description

Successive over-relaxation (SOR) is a generalization of Gauss-Seidel method. Gregory R. Andrews book ([And99, p.546]) explains clearly this method and its parallel implementations.

This benchmark performs an array access intensive test which computes 100 iterations of successive over-relaxation on an $N \times N$ grid. Performance units are iterations per second.

Algorithm methods

The two classes of this algorithm are `JGFSORBench` and `SOR`. `JGFKernel` method in class `JGFSORBench` is the responsible for the data partition and distribution. It basically splits the matrix in blocks with the last two rows replicated, as represented in figure 18. Each processor stores its block in array `p_G` and the process rank corresponds to the block number. Code 15 shows how data partition is implemented.

`SORrun` method in class `SOR` is the SOR algorithm that will run on each processor with different `p_G` arrays. It uses a "red-black" ordering mechanism [And99, p.546] to simplify its parallelisation.

Algorithm parallelization

In order to update elements of the principle array during each iteration, neighbouring elements of the array are required, including elements previously updated. Hence this benchmark is, in this form, inherently serial. To allow parallelisation to be carried out, the algorithm

Code 15 SOR algorithm: data partition

```
87   /* copy or send the values of G to the sub arrays p_G */
88   if(rank==0) {
89       if(nprocess==1) {
90           iup = p_row+1;
91       } else {
92           iup = p_row+2;
93       }
94
95       for(int i=1;i<iup;i++) {
96           for(int j=0;j<p_G[0].length;j++) {
97               p_G[i][j] = G[i-1][j];
98           }
99       }
100
101      for(int j=0;j<G[0].length;j++) {
102          p_G[0][j] = 0.0;
103      }
104
105      for(int k=1;k<nprocess;k++) {
106          if(k==nprocess-1) {
107              m_length = rem_p_row + 1;
108          } else {
109              m_length = p_row + 2;
110          }
111          MPI.COMM_WORLD.Send(G,(k*p_row)-1,m_length,MPI.OBJECT,k,k);
112      }
113
114  } else {
115      MPI.COMM_WORLD.Recv(p_G,0,p_row+2,MPI.OBJECT,0,rank);
116  }
117
118  if(rank==(nprocess-1)) {
119      for(int j=0;j<datasizes[size];j++){
120          p_G[p_G.length-1][j] = 0.0;
121      }
122  }
```

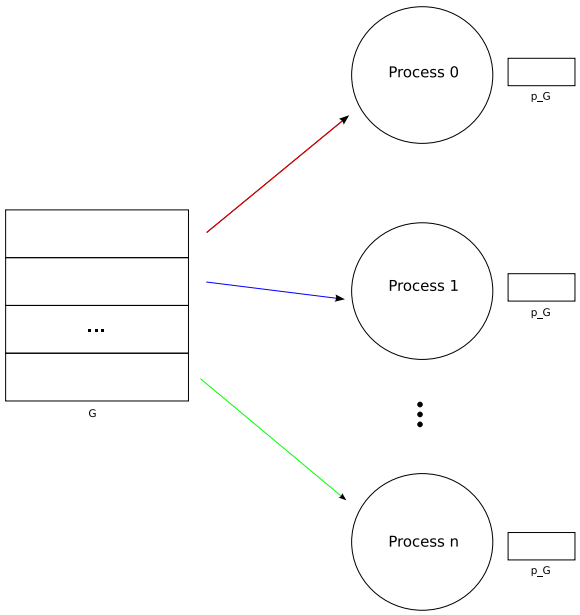


Figure 18: Data distribution in SOR algorithm

uses a "red-black" ordering mechanism. This allows the loop over array rows to be parallelised, hence the outer loop over elements is distributed between processors in a block manner. [SBO01]

Using this strategy, the outer loop runs twice over iterations and in each cycle iteration it works only on even or on odd entries (running the loop twice over iterations assure that all entries are used). Code 16 shows how the loop is programmed.

Code 16 SOR algorithm: main loop

```
56  for (int p=0; p<2*num_iterations; p++) {
57    for (int i=ilow+(p%2); i<ihigh; i=i+2) {
        ...
103   }
      ...
117 }
```

At the end of each iteration, all the processes need to synchronize with their neighbours. All processes, except the last one (`nprocess-1`), exchange data with the next neighbour process (`rank+1`), sending it the ante-penultimate row and receiving from it the last row. In the same way, all processes, except the first one, exchange data with the previous neighbour process (`rank-1`), sending it the second row and receiving from it the first row.

Figure 19 illustrates this synchronization mechanism and Code 17 show how it is programmed.

Code 17 SOR algorithm: synchronization between processes

```
107  if(JGFSORBench.rank!=JGFSORBench.nprocess-1) {
108    MPI.COMM_WORLD.Sendrecv(p_G[p_G.length-2],0,
                               p_G[p_G.length-2].length,
                               MPI.DOUBLE,
109                            JGFSORBench.rank+1,1,
110                            p_G[p_G.length-1],0,
                               p_G[p_G.length-1].length,MPI.DOUBLE,
                               JGFSORBench.rank+1,2);
111  }
112  if(JGFSORBench.rank!=0){
113    MPI.COMM_WORLD.Sendrecv(p_G[1],0,p_G[1].length,
                               MPI.DOUBLE,JGFSORBench.rank-1,2,
114                            p_G[0],0,p_G[0].length,MPI.DOUBLE,
                               JGFSORBench.rank-1,1);
115  }
```

When we reach the desired number of iterations and the last synchronization step between processes, every process sends to process rank 0 its partial results (stored in array `p_G`) and process rank 0 joins them in the result matrix `G`. Code 18 shows how this step is programmed.

The process interaction model used by this algorithm is usually called **Heartbeat**, because data is divided among workers, each is responsible for updating a particular part and new data values depend on values held by worker on their immediate neighbors. It is called

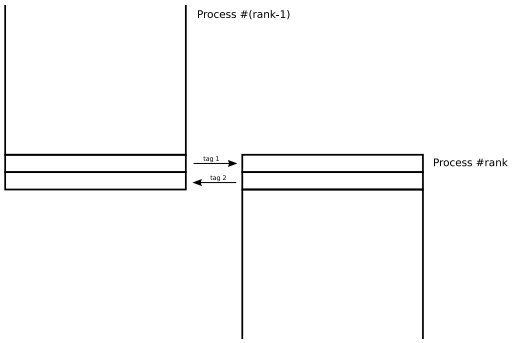


Figure 19: Synchronization mechanism in SOR algorithm

Code 18 SOR algorithm: data merging

```
123  MPI.COMM_WORLD.Barrier();
124  System.gc();
125  if(JGFSORBench.rank==0) {

127    for(int i=1;i<p_G.length-1;i++) {
128      for(int j=0;j<G[0].length;j++) {
129        G[i-1][j] = p_G[i][j];
130      }
131    }

133    for(int k=1;k<JGFSORBench.nprocess;k++) {
134      if(k==(JGFSORBench.nprocess-1)) {
135        rm_length = JGFSORBench.rem_p_row;
136      } else {
137        rm_length = JGFSORBench.p_row;
138      }
139      MPI.COMM_WORLD.Recv(G,k*JGFSORBench.p_row,rm_length,MPI.OBJECT,k,k);
140      System.gc();
141    }

144  } else {

146    for(int k=1;k<JGFSORBench.nprocess;k++){
147      if(JGFSORBench.rank==k) {
148        MPI.COMM_WORLD.Ssend(p_G,1,JGFSORBench.p_row,
                                MPI.OBJECT,0,JGFSORBench.rank);
149      }
150    }
151  }
```

Heartbeat because the actions of each worker are like the beating of a heart: expand, sending information out; contract, gathering new information; then process the information and repeat [And99].

A.1.4. Crypt: IDEA encryption

Algorithm description

In cryptography, the International Data Encryption Algorithm (IDEA) is a block cipher designed by Xuejia Lai and James L. Massey of ETH-Zürich and was first described in 1991. The algorithm was intended as a replacement for the Data Encryption Standard. IDEA is a minor revision of an earlier cipher, PES (Proposed Encryption Standard); IDEA was originally called IPES (Improved PES).

This benchmark test performs IDEA encryption then decryption and it is based on code presented in Applied Cryptography by Bruce Schneier, which was based on code developed by Xuejia Lai and James L. Massey. Performance units are iterations per second.

Algorithm methods

This algorithm's main class is `IDEATest` and its control class is `JGFCryptBench`. The control class determines the array size (`p_array_rows`) on each process, as Code 19 illustrates.

Code 19 Crypt algorithm: partial arrays initialization

```
42  public void JGFinitialise(){
43    array_rows = datasizes[size];
44
45    /* determine the array dimension size on each process
46       p_array_rows will be smaller on process (nprocess-1).
47       ref_p_array_rows is the size on all processes except process (nprocess-1),
48       rem_p_array_rows is the size on process (nprocess-1).
49    */
50
51    p_array_rows = (((array_rows / 8) + nprocess -1) / nprocess)*8;
52    ref_p_array_rows = p_array_rows;
53    rem_p_array_rows = p_array_rows - ((p_array_rows*nprocess) - array_rows);
54    if(rank==(nprocess-1)){
55      if((p_array_rows*(rank+1)) > array_rows) {
56        p_array_rows = rem_p_array_rows;
57      }
58    }
59
60    buildTestData();
61  }
```

`buildTestData` method, illustrated in Code 20, creates on process rank 0 three byte arrays (`plain1`, `crypt1` and `plain2`). It also creates three smaller byte arrays on every process (`p_plain1`, `p_crypt1` and `p_plain2`) that will be used for partial results. The size of these arrays was determined in the data initialisation in class `JGFCryptBench`.

The main method of this test is `cipher_idea`, which processes plaintext in 64-bit blocks, one at a time, breaking the block into four 16-bit unsigned subblocks. It goes through eight rounds of processing using 6 new subkeys each time, plus four for last step. The source text is in array `p_plain1`, the destination text goes into array `p_plain2` The routine represents 16-bit subblocks and subkeys as type `int` so that they can be treated more easily as unsigned. Multiplication modulo 0x10001 interprets a zero sub-block as 0x10000; it must to fit in 16 bits.

`Do` method is the one which calls `cipher_idea` to crypt and decrypt the message.

Algorithm parallelization

Data distribution is made in the method `Do`, where process rank 0 splits the plain message and send the tagged pieces (using `Ssend` primitive) to the other processes. Code 21 shows how data distribution is implemented and Figure 20 illustrates it. Note that process rank 0 has a `p_plain1` array too, but it doesn't need to be broadcasted (figure 20 marks `p_plain1` in process 0 as *local*).

Code 20 Crypt algorithm: data initialization

```
129  void buildTestData()
130  {
131
132
133    // Create three byte arrays that will be used (and reused) for
134    // encryption/decryption operations.
135
136    if(JGFCryptBench.rank==0) {
137      plain1 = new byte [array_rows];
138      crypt1 = new byte [array_rows];
139      plain2 = new byte [array_rows];
140    }
141
142    p_plain1 = new byte [p_array_rows];
143    p_crypt1 = new byte [p_array_rows];
144    p_plain2 = new byte [p_array_rows];
145
146    Random rndnum = new Random(136506717L);  // Create random number generator.

...

159    userkey = new short [8];  // User key has 8 16-bit shorts.
160    Z = new int [52];         // Encryption subkey (user key derived).
161    DK = new int [52];        // Decryption subkey (user key derived).

...

165    for (int i = 0; i < 8; i++)
166    {
167        // Again, the random number function returns int. Converting
168        // to a short type preserves the bit pattern in the lower 16
169        // bits of the int and discards the rest.
170
171      userkey[i] = (short) rndnum.nextInt();
172    }
173
174    // Compute encryption and decryption subkeys.
175
176    calcEncryptKey();
177    calcDecryptKey();
178
179    // Fill plain1 with "text."
180    // do on process 0 for reference
181
182    if(JGFCryptBench.rank==0) {
183      for (int i = 0; i < array_rows; i++)
184      {
185        plain1[i] = (byte) i;
186
187        // Converting to a byte
188        // type preserves the bit pattern in the lower 8 bits of the
189        // int and discards the rest.
190      }
191    }
192
193  }
```

Code 21 Crypt algorithm: data distribution

```
82  if(JGFCryptBench.rank==0) {
83    for (int i = 0; i < p_array_rows; i++) {
84      p_plain1[i] = plain1[i];
85    }
86    for(int k=1;k<JGFCryptBench.nprocess;k++){
87      if(k==JGFCryptBench.nprocess-1) {
88        m_length = rem_p_array_rows;
89      } else {
90        m_length = p_array_rows;
91      }
92      MPI.COMM_WORLD.Ssend(plain1,(p_array_rows*k),m_length,MPI.BYTE,k,k);
93     }
94   } else {
95     MPI.COMM_WORLD.Recv(p_plain1,0,p_array_rows,MPI.BYTE,0,JGFCryptBench.rank);
96   }
```

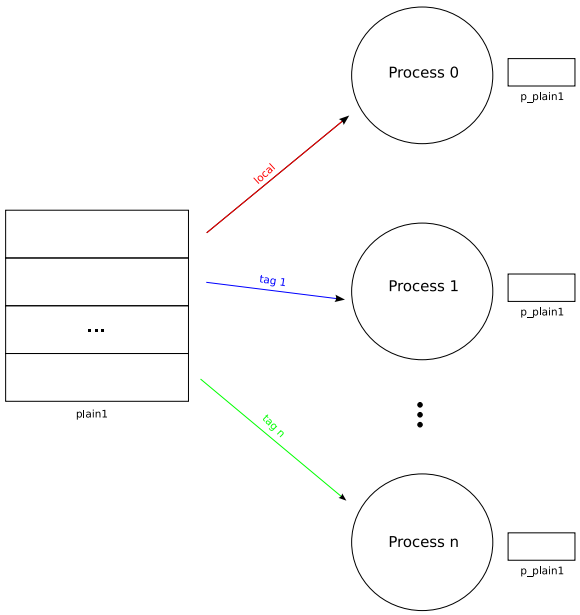


Figure 20: Data distribution in IDEA encryption algorithm

Each process receives its piece, encrypts it, decrypts it and then send the result (`p_plain1`) back to process rank 0 which puts all the pieces together. Code 22 illustrates this process.

Code 22 Crypt algorithm: processing and data merging

```
98   cipher_idea(p_plain1, p_crypt1, Z);     // Encrypt plain1.
99   cipher_idea(p_crypt1, p_plain2, DK);    // Decrypt.
100
101  MPI.COMM_WORLD.Barrier();
102
103  if(JGFCryptBench.rank==0) {
104    for(int k=0; k<p_array_rows;k++){
105      plain2[k] = p_plain2[k];
106    }
107
108    for(int k=1;k<JGFCryptBench.nprocess;k++) {
107      MPI.COMM_WORLD.Recv(plain2,(p_array_rows*k),p_array_rows,MPI.BYTE,k,k);
110    }
111  } else {
112    MPI.COMM_WORLD.Ssend(p_plain2,0,p_array_rows,MPI.BYTE,0,JGFCryptBench.rank);
113  }
```

The process interaction model used by this algorithm is usually called **Farming**.

A.1.5. Sparse Matrix Multiplication

Algorithm description

The matrix product $A \times B = C$, where A and B are $n \times n$ matrices requires n^2 inner products. Each inner product is the sum (plus reduction) of the pairwise products of two vectors of length n.

A sparse matrix is a matrix where most entries are zero and the product of two sparse matrices is also a sparse matrix. This is because a zero in the two argument matrices will lead to a zero in n vector products.

This test multiplies a $N \times N$ sparse matrix stored in compressed-row format by a dense vector 10 times. Performance units are iterations per second.

The compressed row format (CRS) puts the subsequent nonzeros of the matrix rows in contiguous memory locations. Assuming we have a nonsymmetric sparse matrix A, we create three vectors: one for floating point numbers (val) and the other two for integers (col_ind, row_ptr). The val vector stores the values of the nonzero elements of the matrix A as they are traversed in a row-wise fashion. The col_ind vector stores the column indexes of the elements in the val vector. That is, if $val(k) = a_{ij}$, then $col\_ind(k) = j$. The row_ptr vector stores the locations in the val vector that start a row; that is, if $val(k) = a_{ij}$, then $row\_ptr(i) =< k < row\_ptr(i+1)$.

Algorithm methods

This algorithm's main class is `SparseMatmult` and its control class is `JGFSparseMatmultBench`. The two main methods are `JGFinitialise()` (class `JGFSparseMatmultBench`) and `test()` (class `SparseMatmult`).

This is a case where the control class also distributes data among the processes. In the method `JGFinitialise()`, process rank 0 creates the three vectors that represent the sparse matrix (`buf_val`, `buf_row` and `buf_col`). It also creates three partial vectors with random values (`val`, `row` and `col`) which will be used by the other processes. These three arrays will have different values for different processes. Code 23 illustrates this process.

Code 23 Sparse matrix multiplication: data initialization

```
64  public void JGFinitialise() throws MPIException{
65
66
67    /* Determine the size of the arrays row,val and col on each
68       process. Note that the array size on process (nprocess-1) may
69       be smaller than the other array sizes.
70    */
71
72  p_datasizes_nz = (datasizes_nz[size] + nprocess -1) /nprocess;
73  ref_p_datasizes_nz = p_datasizes_nz;
74  rem_p_datasizes_nz = p_datasizes_nz - ((p_datasizes_nz*nprocess) - datasizes_nz[size]);
75  if(rank==(nprocess-1)){
76   if((p_datasizes_nz*(rank+1)) > datasizes_nz[size]) {
77     p_datasizes_nz = rem_p_datasizes_nz;
78   }
79  }
80
81  /* Initialise the arrays val,col,row. Create full sizes arrays on process 0 */
82
83  x = RandomVector(datasizes_N[size], R);
84  y = new double[datasizes_M[size]];
85  p_y = new double[datasizes_M[size]];
86
87  val = new double[p_datasizes_nz];
88  col = new int[p_datasizes_nz];
89  row = new int[p_datasizes_nz];
90
91  if(rank==0) {
92    buf_val = new double[datasizes_nz[size]];
93    buf_col = new int[datasizes_nz[size]];
94    buf_row = new int[datasizes_nz[size]];
95  }
```

In the method `test()`, array `p_y` is calculated based on the three partial arrays `val`, `row` and `col`, which means that different processes will calculate different parts of `p_y`.

Algorithm parallelization

Data distribution is made in the method `JGFinitialise` which sends to each process parts of the three complete arrays, as shown in Code 24. Figure 21 illustrates the distribution of one of those vectors - `buf_val`. Each process stores the received values in arrays `row`, `col` and `val` and calculates different parts of the matrix.

Code 24 Sparse matrix multiplication: data distribution

```
101   if(rank==0) {
102
103    for (int i=0; i<p_datasizes_nz; i++) {
104
105        // generate random row index (0, M-1)
106        row[i] = Math.abs(R.nextInt()) % datasizes_M[size];
107        buf_row[i] = row[i];
108        // generate random column index (0, N-1)
109        col[i] = Math.abs(R.nextInt()) % datasizes_N[size];
110        buf_col[i] = col[i];
111        val[i] = R.nextDouble();
112        buf_val[i] = val[i];
113    }
114
115    for(int k=1;k<nprocess;k++) {
116      if(k==nprocess-1) {
117        p_datasizes_nz = rem_p_datasizes_nz;
118      }
119      for (int i=0; i<p_datasizes_nz; i++) {
120        buf_row[i+(k*ref_p_datasizes_nz)] = Math.abs(R.nextInt()) % datasizes_M[size];
121        buf_col[i+(k*ref_p_datasizes_nz)] = Math.abs(R.nextInt()) % datasizes_N[size];
122        buf_val[i+(k*ref_p_datasizes_nz)] = R.nextDouble();
123      }
124      MPI.COMM_WORLD.Ssend(buf_row,(k*ref_p_datasizes_nz),p_datasizes_nz,MPI.INT,k,1);
125      MPI.COMM_WORLD.Ssend(buf_col,(k*ref_p_datasizes_nz),p_datasizes_nz,MPI.INT,k,2);
126      MPI.COMM_WORLD.Ssend(buf_val,(k*ref_p_datasizes_nz),p_datasizes_nz,MPI.DOUBLE,k,3);
127    }
128
129    p_datasizes_nz = ref_p_datasizes_nz;
130  } else {
131    MPI.COMM_WORLD.Recv(row,0,p_datasizes_nz,MPI.INT,0,1);
132    MPI.COMM_WORLD.Recv(col,0,p_datasizes_nz,MPI.INT,0,2);
133    MPI.COMM_WORLD.Recv(val,0,p_datasizes_nz,MPI.DOUBLE,0,3);
134  }
```

For the $(i+1)^{th}$ iteration, each process will need the values calculated from other processes in iteration i. `AllReduce` primitive is used at the end of every iteration to update the values on each process. `AllReduce` primitive is the same as reduce except that the result appears in receive buffer of all process in the group. Code 25 illustrates the use of the `AllReduce` primitive.

The process interaction model used by this algorithm without iterations is usually called Farming, but since it is iterative and it needs to synchronize at the end of each iteration, the model is the **Heartbeat**.

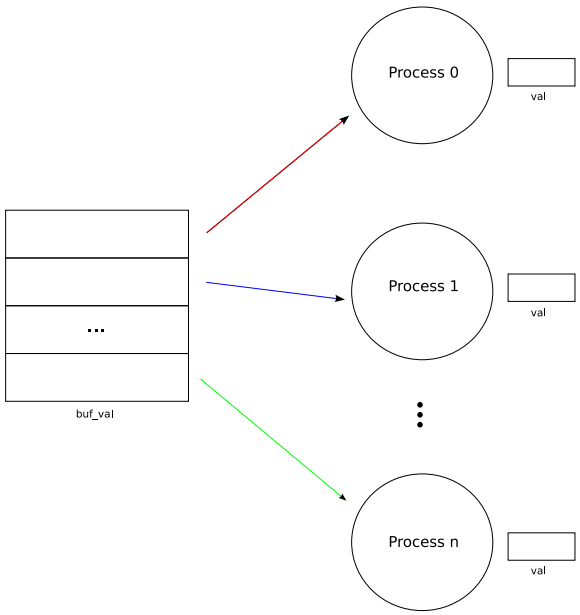


Figure 21: Data distribution in Sparse algorithm

Code 25 Sparse matrix multiplication: intermediate communication and synchronization

```
40   MPI.COMM_WORLD.Barrier();
41   if(JGFSparseMatmultBench.rank==0){
42     JGFInstrumentor.startTimer("Section2:SparseMatmult:Kernel");
43   }
44
45   for (int reps=0; reps<NUM_ITERATIONS; reps++)
46   {
47       for (int i=0; i<nz; i++)
48       {
49           p_y[ row[i] ] += x[ col[i] ] * val[i];
50       }
51           // create updated copy on each process
52   MPI.COMM_WORLD.Allreduce(p_y,0,y,0,y.length,MPI.DOUBLE,MPI.SUM);
53   }
54
55   MPI.COMM_WORLD.Barrier();
```

A.2. Section 3: Large Scale Applications

A.2.1. Molecular Dynamics simulation

Algorithm description

MolDyn is an N-body code modelling particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. Performance is reported in interactions per second. The number of particles is give by N. The original Fortran 77 code was written by Dieter Heerman, Institut für Theoretische Physik, Germany and converted to Java by Lorna Smith, EPCC.

Algorithm diagrams

The main method of this algorithm is `runiters` (in class `md`), where all the parallelism and communication is performed and where the forces on particles are calculated. There is other class (in the same file) named `particle` which contains methods that work on particles.

Algorithm parallelization

The computationally intense component of the benchmark is the force calculation, which calculates the force on a particle in a pair wise manner. This involves an outer loop over all particles in the system and an inner loop ranging from the current particle number to the total number of particles. The outer loop has been parallelised by dividing the range of the iterations of the outer loop between the processes, in a cyclic manner to avoid load imbalance [SBO01]. A copy of the all the particle data is maintained on each process.

The outer loop is in method `runiters` (class `md`, line 217) and the inner loop is in method `force` (class `particle`, line 366).

The data distribution made in method `runiters` is illustrated in figure 22 and Code 26 shows how it is programmed.

Code 26 Molecular Dynamics simulation: data distribution

```
231  for (i=0+JGFMolDynBench.rank;i<mdsize;i+=JGFMolDynBench.nprocess) {
232      one[i].force(side,rcoff,mdsize,i);  /* compute forces */
233  }
```

`runiters` method also makes a global reduction on partial sums of the forces, epot, vir and interactions. It uses `AllReduce` primitive with the operation `MPI.SUM` and with arrays `tmp_xforce`, `tmp_yforce`, `tmp_zforce`, `tmp_epot`, `tmp_vir` and `tmp_interactions`. Code 27 shows how the global reduction is implemented.

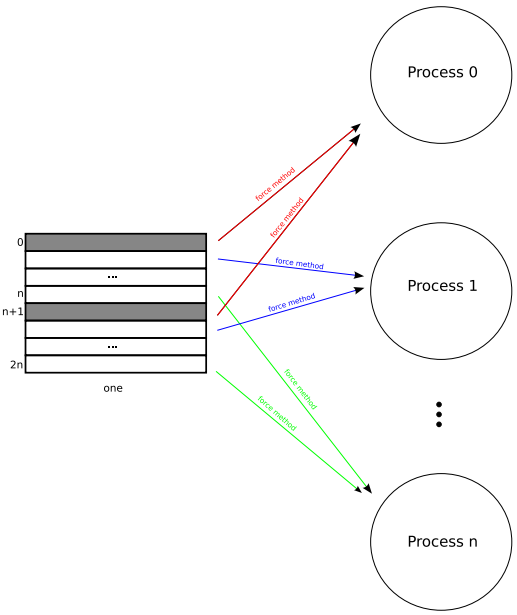


Figure 22: Data distribution in molecular dynamics simulation

Code 27 Molecular Dynamics simulation: global reduction

```
237  /* global reduction on partial sums of the forces,
        epot, vir and interactions */


240      for (i=0;i<mdsize;i++) {
241        tmp_xforce[i] = one[i].xforce;
242        tmp_yforce[i] = one[i].yforce;
243        tmp_zforce[i] = one[i].zforce;
244      }

246      MPI.COMM_WORLD.Allreduce(tmp_xforce,0,tmp_xforce,
                                  0,mdsize,MPI.DOUBLE,MPI.SUM);
247      MPI.COMM_WORLD.Allreduce(tmp_yforce,0,tmp_yforce,
                                  0,mdsize,MPI.DOUBLE,MPI.SUM);
248      MPI.COMM_WORLD.Allreduce(tmp_zforce,0,tmp_zforce,
                                  0,mdsize,MPI.DOUBLE,MPI.SUM);

250      for (i=0;i<mdsize;i++) {
251        one[i].xforce = tmp_xforce[i];
252        one[i].yforce = tmp_yforce[i];
253        one[i].zforce = tmp_zforce[i];
254      }

256      tmp_epot[0] = epot;
257      tmp_vir[0] = vir;
258      tmp_interactions[0] = interactions;

260      MPI.COMM_WORLD.Allreduce(tmp_epot,0,tmp_epot,
                                  0,1,MPI.DOUBLE,MPI.SUM);
261      MPI.COMM_WORLD.Allreduce(tmp_vir,0,tmp_vir,
                                  0,1,MPI.DOUBLE,MPI.SUM);
262      MPI.COMM_WORLD.Allreduce(tmp_interactions,0,tmp_interactions,
                                  0,1,MPI.DOUBLE,MPI.SUM);

264      epot = tmp_epot[0];
265      vir = tmp_vir[0];
266      interactions = tmp_interactions[0];

268      MPI.COMM_WORLD.Barrier();
```

A.2.2. Monte Carlo simulation

Algorithm description

A simulation can be seen as a method meant to imitate a real-life system. A Monte Carlo simulation is a simulation where random values are generated for some (or all) variables over and over.

This test performs a financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset. The code generates N sample time series with the same mean and fluctuation as a series of historical data. Performance is measured in samples per second.

Algorithm methods

This algorithm's main classes are `AppDemo` and `PriceStock`. Class `AppDemo` defines method `runSerial`, where all the parallelism and communication is performed and where the `run` method from class `PriceStock` is called. The `run` method will generate a single sequence with the required statistics, estimate its volatility, expected return rate and final stock price value.

Algorithm parallelization

The principle loop over number of Monte Carlo runs can be easily parallelised by dividing the work in a block fashion.

Each process will execute `run` method (from class `PriceStock`) with different tasks as argument, generating a single sequence with the required statistics and estimating its volatility, expected return rate and final stock price value. The result of the computation will be stored in `p_results` vector which will be sent to process rank 0.

Process rank 0 receives all the results and stores them in `results` vector.

Code 28 illustrates the main parallelization code.

The process interaction model used by this algorithm is usually called **Farming**.

Code 28 Monte Carlo Simulation: algorithm parallelization

```
174  ilow = JGFMonteCarloBench.rank*p_nRunsMC;
175  ihigh = (JGFMonteCarloBench.rank+1)*p_nRunsMC;
176  if (JGFMonteCarloBench.rank==JGFMonteCarloBench.nprocess-1) ihigh = nRunsMC;
177
178  // Now do the computation.
179  PriceStock ps;
180  for( int iRun=ilow; iRun < ihigh; iRun++ ) {
181    ps = new PriceStock();
182    ps.setInitAllTasks(initAllTasks);
183    ps.setTask(tasks.elementAt(iRun));
184    ps.run();
185    p_results[0].addElement(ps.getResult());
186  }
187
188  if(JGFMonteCarloBench.rank==0) {
189    for(int i=0;i<p_results[0].size();i++){
190      results.addElement((ToResult) p_results[0].elementAt(i));
191    }
192    for(int j=1;j<JGFMonteCarloBench.nprocess;j++) {
193      p_results[0].removeAllElements();
194      MPI.COMM_WORLD.Recv(p_results,0,1,MPI.OBJECT,j,j);
195      for(int i=0;i<p_results[0].size();i++){
196        results.addElement((ToResult) p_results[0].elementAt(i));
197      }
198
199    }
200
201  } else {
202
203    MPI.COMM_WORLD.Send(p_results,0,1,MPI.OBJECT,0,JGFMonteCarloBench.rank);
204
205  }
```

A.2.3. 3D Ray Tracer

Algorithm description

Ray tracing is a general technique from geometrical optics of studying the path taken by light by following rays of light as they interact with optical surfaces. Its primary applications are in 3D computer graphics, where scenes are rendered by following rays from the eyepoint to light sources, and in the design of optical systems, such as lenses and mirrors.[12]

The JGF benchmark measures the performance of a 3D ray tracer. The rendered scene contains 64 spheres, and is rendered at a resolution of $N \times N$ pixels. Performance units are pixels per second.

Algorithm methods

Raytracer benchmark depends on the following classes:

`JGFRayTracerBench` - this is the control class;

`RayTracer` - this is the main class. The most important method is `render` which receives a `Interval` object and launches a ray of light for each point that belongs to the interval, calling method `trace`. `trace` method calculates light intersections with the scene and then it calls `shade` method to determine the shaded color. `shade` method calls `trace` method to calculate indirect light influence;

`Primitive` - this abstract class defines abstract methods for primitives, like spheres;

`Interval` - this class represents the interval for which the image is rendered; it looks overkill, but it's a good approach to distribute the algorithm;

`Isect` - this class represents intersections;

`Light` - this class represents light points;

`Ray` - this class represents rays of light;

`Scene` - this class represents the scene. It has methods to manipulate the scene objects and lights;

`Sphere` - this class represents spheres; it is subclass of `Primitive`

`Surface` - this class represents surfaces (their color, shine and other properties);

[12] Wikipedia definition

`View` - this class represents view points;

The main method of this algorithm is `render` (in class `RayTracer`, line 186) where all the parallelism and communication is performed.

Algorithm parallelization

The outermost loop (over row of pixels) in the method `render` has been parallelised using a cyclic distribution for load balance. Since each interval point can be calculated independently, each iteration of the loop can be distributed between processes. `Interval` class looks a bad approach for sequential versions, but is very useful for the parallel version.

Code 29 shows the loop responsible for distribution and load balancing.

Code 29 Raytracer: data partition

```
239  for( y =  interval.yfrom+JGFRayTracerBench.rank;
       y <  interval.yto;
       y += JGFRayTracerBench.nprocess) {

        ...

270  }
```

Each process **creates** the scene to be rendered and produces image rows in a cyclic fashion, as figure 23 shows.

All the processes create a `p_row` variable to hold the image rows. This creation is done in the line 193 (Code 30).

Code 30 Raytracer: partial results

```
193  if(JGFRayTracerBench.rank==0){
194      row = new int[interval.width * (interval.yto-interval.yfrom)];
195  }

197  int p_row[] = new int[((((interval.width *
                    (interval.yto-interval.yfrom))
                    /interval.width) +
                    JGFRayTracerBench.nprocess-1) /
                    JGFRayTracerBench.nprocess)*interval.width];
```

Processes different from rank 0 send their computed image rows (`p_row` array) to process rank 0 which merges them in `row` variable. Code 31 shows how this behaviour is programmed.

Each process calculates a `checksum` value, which is used to validate the algorithm. `AllReduce` primitive is used to update this value making a global sum on checksum, as code 32 shows.

The process interaction model used by this algorithm is usually called **Farming**.

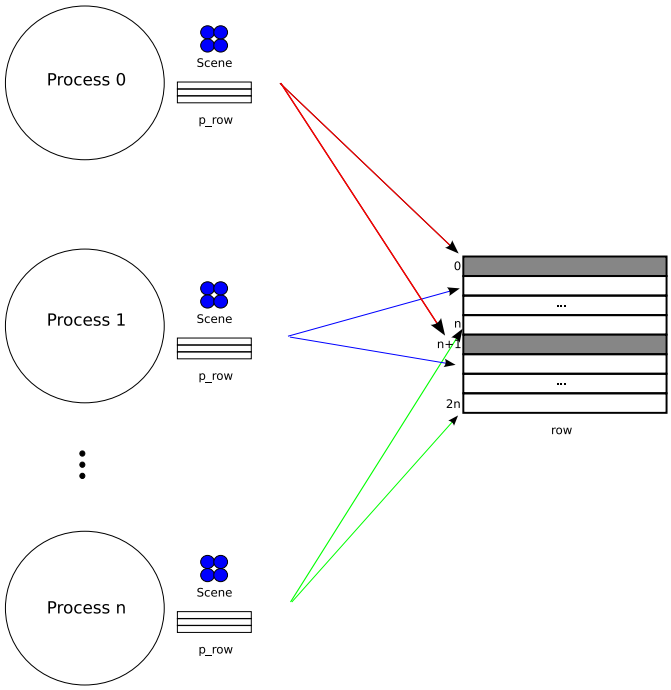


Figure 23: Data distribution and load balancing in raytracer algorithm

Code 31 Raytracer: data merging

```
282  if(JGFRayTracerBench.rank==0) {
283    for(int k=0;k<JGFRayTracerBench.nprocess;k++){
284      if(k!=0){
285        MPI.COMM_WORLD.Recv(p_row,0,p_row.length,MPI.INT,k,k);
286      }
287      t_count = 0;
288      for(int i = k;
             i < (interval.yto-interval.yfrom);
             i+=JGFRayTracerBench.nprocess){
289        for(x = 0; x < interval.width; x++) {
290          row[i*interval.width+x] = p_row[t_count];
291          t_count++;
292        }
293      }
294    }
295  } else {
296    MPI.COMM_WORLD.Send(p_row,0,p_row.length,
                           MPI.INT,0,JGFRayTracerBench.rank);
297  }
```

Code 32 Raytracer: validation

```
274  tmp_checksum[0] = (double) checksum;
275  MPI.COMM_WORLD.Reduce(tmp_checksum,0,tmp_checksum,0,
                           1,MPI.DOUBLE,MPI.SUM,0);
276  if(JGFRayTracerBench.rank==0) {
277    checksum = (long) tmp_checksum[0];
278  }
```

B. Automatic object distribution implementation

B.1. Parser generators and related tools

This appendix describes the parser generators that were analyzed to implement the code transformations explained in section 2.

B.1.1. ANTLR

URL:	http://www.antlr.org
License:	Public domain
Generated Language:	Java
Java Grammar:	Yes (Java 1.4 and 1.5)
Used by:	**Jade** - a parallel message-drive Java; **BlueJ** - an integrated Java environment specifically designed for introductory teaching; **USFJProf** - a Java profiling tool.

B.1.2. JavaCC

URL:	http://javacc.dev.java.net
License:	Berkeley Software Distribution (BSD) License
Generated Language:	Java
Java Grammar:	Yes (Java 1.4 and 1.5)
Used by:	**Xilize** - a tool to help creating XHTML; **JPython** - a tool that allows to run Python on any Java platform.

B.1.3. CUP Parser Generator

URL:	http://www.cs.princeton.edu/~appel/modern/java/CUP/
License:	GPL compatible
Generated Language:	Java
Java Grammar:	Yes (Java 1.5)
Used by:	

B.1.4. JParse

JParse is a parser, type evaluator, and exception analyzer for the Java language. It is a freely redistributable library of Java code. It is based on ANTLR, builds an abstract syntax tree (AST) and contains a tree parser to traverse the created trees.

URL:	http://www.ittc.ku.edu/JParse
License:	LGPL
Java Grammar:	Yes (Java 1.4)

B.2. Frontend script

In order to simplify the source code transformation and the automatic object distribution we have created a script that uses a configuration file where the following variables are defined:

ppc_manager_name - cluster manager name to register in the nameserver;

ppc_manager_host - host where the cluster manager runs;

nameserver_host - host where the nameserver runs;

nameserver_port - port where the nameserver listens;

ppc_nodes - list of nodes where remote objects can be created;

The script is run from command line and can be seen as a frontend for several tools. It provides the following options:

pre <files> - transforms the specified files and generates support classes, as described in Chapter 2;

rmic - generates stub and skeleton class files for remote objects; it determines automatically which are the remote objects;

compile <files> - compiles the files given as argument, defining all the needed parameters;

start - starts the nameserver, the cluster manager and the factories;

stop - stops the nameserver, the cluster manager and the factories;

status - shows the system status (if it is ready to distribute objects);

flags - prints the Java options needed to use the nameserver.