Exploiting implicit loop parallelism using multiple multithreaded servers in Java

Fabian Breg Aart Bik Dennis Gannon

December 16, 1997

1 Introduction

Since its introduction in the late eighties, the global **Internet** has grown from a wide area information repository to a large metacomputer consisting of a great number of high performance computers. To exploit this computing power the heterogeneous nature of the Internet has to be overcome. With the introduction of **Java** [6] and its accompanying **Bytecode** [7], true portable programs can be written in a high level language that can be downloaded to and run on any computer that hosts a Bytecode interpreter. This feature, together with the communication that the API provides, makes Java a suitable language to implement distributed software systems.

The interpretative nature of the Java programs makes it less suitable for high performance computing. Currently several **Just In Time** (JIT) compilers are available, which translate the Java Bytecode to native machine code just prior to execution. Other attempts to make Java suitable for high performance computing consists of optimizing Java compilers and Java restructuring compilers. An example of the latter kind is `Javar` [3], which exploits loop parallelism as well as multi-way recursive method parallelism using the multithreading facilities of Java as described in [2].

This document describes the source code transformations needed to exploit loop parallelism using multiple multithreaded servers in a distributed system. We propose and implement a source code restructuring strategy, which uses multiple servers each, running a number of workers to execute a subset of iterations of parallel loops. The communication among these servers is implemented in Java **Remote Method Invocation** (RMI). Within a server, each worker runs in its own thread of control, which can run in parallel on those machines that supports true multithreading. Figure 1 shows the client, the servers and their workers.

The parallelization is driven by annotations that have to be inserted manually in the source code. We propose a set of annotations which can be used to exploit parallelism in *fully independent* for-loops. Several techniques for **dependency analysis** exist [1, 11], which can be used to detect loop parallelism. Currently, our compiler does not perform such dependency analysis, but instead relies on the annotations to correctly identify a parallel loop. Do-across like loops are not implemented, because the synchronization overhead in a distributed system tends to be too high.

This project extends the work done in [2]. `Javar`, the compiler developed in this project, is extended to include the transformation of Java programs as described in this document.

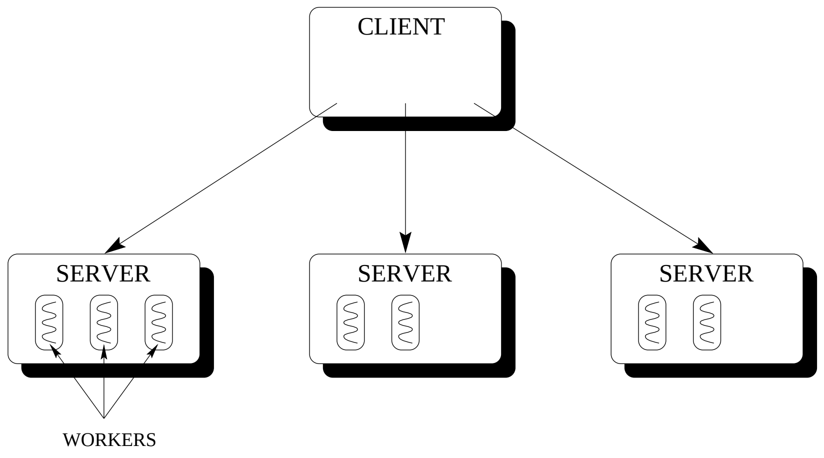


Figure 1: Client, Servers and Workers architecture

The outline of this article is as follows: Section 2 describes Java's multithreading and communication features in some detail. Next, Section 3 describes the syntax and semantics of our proposed annotations. Section 4 provides an overview of the implementation of this project. Section 5 describes the `distribute` package, which contains some classes that implement common routines. We describe the source transformations and the resulting specific client and server code in Section 6. We present some performance measurements in Section 7. We conclude this article in Section 8.

2 Java

Java is an object oriented language supporting single inheritance and method overloading. Its main attractions are its translation to portable Bytecode, which can be downloaded and run on any machine hosting a **Java Virtual Machine**, its **Abstract Windowing Toolkit** in which Graphical User Interfaces (GUIs) can be constructed, its support for concurrent programming by means of threads and its support for distributed computing using sockets or Remote Method Invocation. This project uses its concurrent and distributed features to parallelize loops, so these two features are first described in some detail. For a description of the Java language we refer [6, 5].

2.1 Java threads

Java has language support for multithreaded execution of programs. The Java API provides classes and interfaces to create and manipulate threads and methods to perform communication between them. Java provides constructs for synchronization in the form of monitors. This section describes the multithreading facilities used in this project. For a more elaborate introduction in Java's multithreading features we refer to [9, 4].

A thread is created by creating an object of the `Thread` class or a subclass of this class. After creation, the thread can be started by calling the `start()` method on the thread, which will create another thread of control executing the `run()` method of the `Thread` class. The thread stops its execution when it reaches the end of the `run()` method or when the `stop()` method is invoked, either from the thread itself of from another thread. A thread that has created another thread can wait for it to terminate by invoking `join()`.

Synchronization between threads is implemented by specifying a method to be `synchronized`, which prevents the method to be run by two threads at the same time. When entering such method, the thread obtains a lock on the object to which the method belongs. When another thread attempts to enter the method it has to wait until the lock is released.

A thread can invoke `wait()` from within synchronized code to wait for a signal from another thread. Invoking `wait()` will release the acquired lock, allowing other threads to execute the synchronized method. A thread can invoke `notify()` or `notifyAll()` to signal one randomly chosen thread or to signal all waiting threads respectively.

2.2 Java Remote Method Invocation

To provide communication between objects in different JVM's, the Java API contains classes that implement the socket mechanism. Although the Java API provides a simple interface for programming sockets, applications still have to implement a protocol for encoding and decoding messages, which is a cumbersome and error-prone task.

Java RMI [10] is designed to simplify the communication between two objects on separate machines by allowing an object to invoke the methods of an object on a remote machine in the same way as methods on local objects are invoked. To invoke methods of a **remote object**, a **remote reference** to that object has to be obtained. Since the object resides in a different name-space, a **registry** is used to manage remote references; RMI servers can register remote objects at the registry after which clients can obtain a reference to these remote objects. We will now briefly describe how Java RMI works. For a more elaborate description we refer to [9].

An overview of RMI is shown in Figure 2. First, a **server** creates a remote object and registers it at the registry, which is represented by arrows (1) in the figure. The client can obtain references to objects stored in the registry as shown by arrows (2). When the client invokes a method on a remote object, the method is actually invoked on a **stub** object, located on the same JVM, instead (3). This stub object makes a message containing the name of the method together with its parameters, a process called **marshalling**, and sends this message to the associated **skeleton** object residing on the server host (4). The skeleton object extracts the method name and parameters from the message, a process called **unmarshalling**, and invokes the appropriate method on the remote object with which it is associated (5). The remote object executes the method and passes the return value back to the skeleton (6). The skeleton in its turn marshals the return value in a message and sends this message to the stub object (7). The stub unmarshals the return value from the message and returns this value to the client program (8).

The methods of a remote object that can be invoked remotely must be specified in an interface that extends the `Remote` interface, which itself is an empty interface. Every method in the interface must be declared to throw the `RemoteException` in order to account for errors during the remote method invocation; a number of subclasses of RemoteException exist which represent the various errors that can occur.

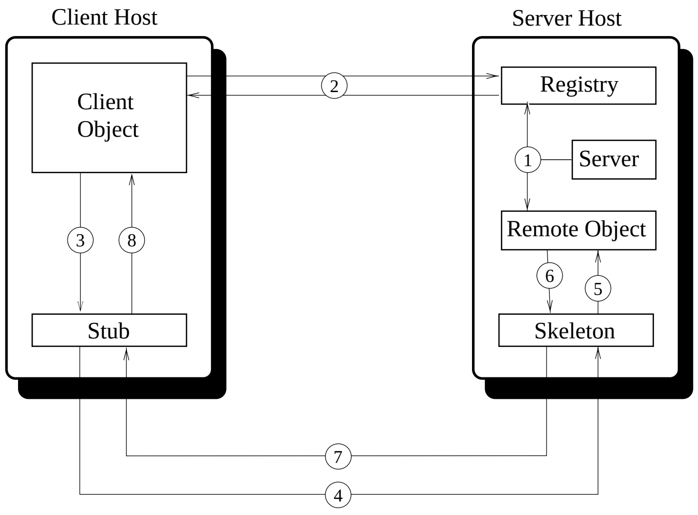
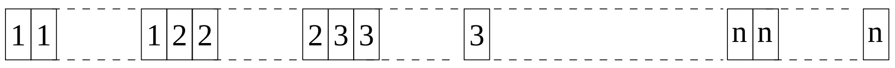


Figure 2: RMI overview

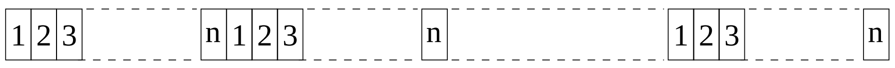
A remote object should be declared to implement at least one remote interface. Furthermore, a remote object can, but need not, extend the `RemoteObject` class or one of its subclasses. Typically, the `UnicastRemoteObject` class is extended by remote objects. After creating a remote object, it has to be exported; if the remote object is declared to extend `UnicastRemoteObject` the object is automatically exported during creation, otherwise it must be explicitly exported using the static method `exportObject()` of the `UnicastRemoteObject` class, providing the remote object as parameter. Once created and exported, a remote object must be registered using either the `bind()` or `rebind()` method of the `Naming` class, providing the remote object and an appropriate name for it.

Creating, exporting and registering remote objects is done by a server object. The server can either use an existing registry or create a registry itself using the `createRegistry()` method of the `LocateRegistry` class, providing the port number to which the registry should listen. To ensure security, the server must install an RMI specific security manager using the `setSecurityManager()` method providing an instance of the `RMISecurityManager` class.

The remote interface of a remote object is used as the type of the remote reference at the client side. A remote reference can be obtained by invoking the `lookup` method of the Naming class, providing the name of the desired remote object. When invoking a method through the remote interface, the `RemoteException` must be caught to handle errors during the remote method invocation. Exceptions that occur during the actual execution of the remote method at the server are thrown to the client encapsulated in a `ServerRuntimeException`. Other exceptions occurring at the server are encapsulated in a `ServerException` object.



(a) block scheduling



(b) cyclic scheduling

Figure 3: Scheduling methods

3 Annotations

To guide the parallelization of Java programs, the user needs to provide annotations in the source code. Therefore, the user needs to identify parallel loops and has to indicate how many servers and workers must be used to execute the parallel loop. In addition, the user has to indicate how the data should be distributed over the hosts. This section describes the syntax and semantics of the annotations.

3.1 Annotating parallel loops

To enforce the generation of parallel code for a loop, its definition must be preceded by the annotation `/*dist*/` (which is ignored by other compilers):

```
/*dist*/
for(int i = 0; i < 10; i++)
    body(i);
```

Optionally the following variables can be specified between the `/*dist` and the `*/`:

- '`<scheduling method>`': This option specifies how loop iterations are assigned to workers. Currently, only `block` scheduling is implemented, in which each worker gets a block of consecutive iterations. A `cyclic` scheduling method, in which consecutive iterations are assigned to different workers, may be added in the future. Figure 3 compares the two scheduling methods. The blocks in this figure represent loop iterations and the numbers in the blocks refer to the worker which executes the iteration.

- '`hosts = <n>`': specifies the number of hosts to be allocated. This value defaults to 2. Note that every host can run multiple workers.

- '`workers = <int>`': specifies the maximum number of workers to start on each server. Together with the number of servers this variable determines the number of jobs to be generated for a parallel loop. If the number of iterations is not a multiple of the number of workers, the actual number of iterations performed by some workers may actually be less than this number.

The following example generates code that can run on two multiprocessor machines with four processors each:

```
/*dist block,
  hosts = 2, threads = 4
*/
for(int i = 0; i < 100; i++)
    doWork(i);
```

The rest of the options that can be specified are concerned with the way in which the data, that is used in the loop, has to be distributed. The next section deals with these annotations.

3.2 Annotating data distribution

To allow workers to access their data efficiently, the compiler needs to be told where to place each piece of data used in the body of the parallel loop. To explain the annotations used for data distribution, a classification of data needs to be made.

The first distinction is between data that is accessed by every worker and data that is only accessed by one particular worker. Data shared among all workers is broadcasted to every server once, while private data is only sent to the specific worker. Note that workers are implemented as threads, so data that is global to the server is automatically accessible by all threads.

A second distinction is between data that is read but not modified (`unmod`) and data that is modified in the body of the loop. The latter kind of data can be subdivided according to whether the modifications of the data are needed after loop execution (`mod_inout`) or are not needed after loop execution (`mod_in`). These kinds of data need and need not be copied back to the client after loop execution, respectively. Note that both the latter kind of data and unmodified data need not be copied back to the client. The difference, however, between the two is caused by the fact that only one copy of global unmodified data is needed, while each worker needs a copy of global modifiable data.

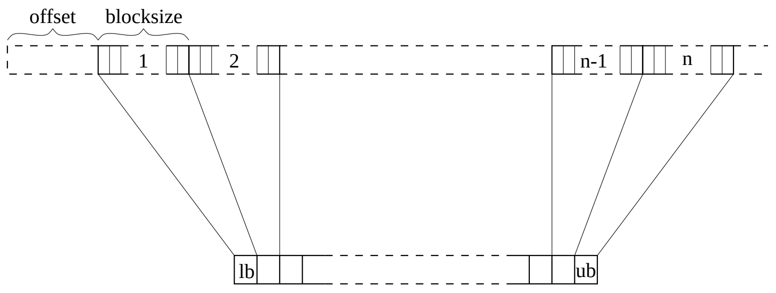
`Mod_out` would correspond to data that need not be copied to the server prior to loop execution. This would mean that this data item is declared outside the loop, without initialization, which happens within the loop. In this case the data cannot be used after the loop, since the compiler cannot make sure that it is indeed initialized after the loop. Thus, such data really must be initialized, in which case it has to be annotate `mod_in` of `mod_inout`.

Using an orthogonal approach, this yields a total of six kinds of data. Data that is global to all workers and which has to be copied back to client after loop execution gives rise to an **output dependence** [11] on that data item, which prevents loop parallelization. This kind of data is therefore not considered further.

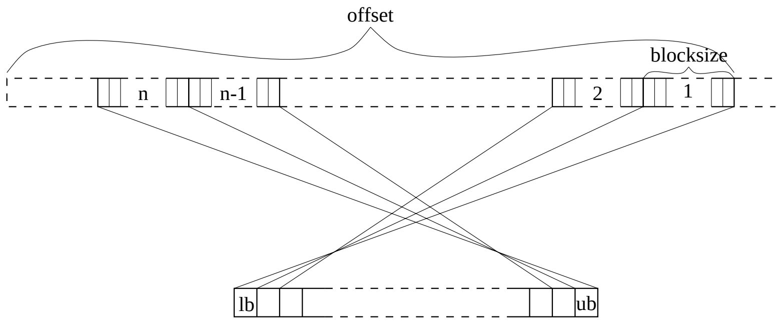
Data global to all workers can be annotated as follows:

- `<varname> <access>`, where access is either `unmod` or `mod_in`.

Only array types can be annotated to be worker private data, since its subparts should all have the same type in order to keep the workers homogeneous. The following annotation is used for worker private data:



(a) Block distribution



(b) Cyclic distribution

Figure 4: Data distribution methods

- `<varname> [offset, blocksize] <distribution> <access>`, where `access` has either the value `unmod`, `mod_in` or `mod_inout`. The `distribution`, `offset` and `blocksize` indicate how the array is divided among the workers as explained below.

For the compiler to be able to split an array among the workers in such a way that every worker has access to those elements from the array that are used in its set of iterations, the access pattern of the array needs to be specified. To be able to capture this pattern in a formula, we only handle fairly simple access patterns. Currently, only blockwise data distribution is implemented as shown in Figure 4.

Figure 4a shows the iterations of a loop in the lower rectangle and the elements of an array in the upper rectangle. In this case, the elements of the array are accessed in a blockwise fashion from left to right. In Figure 4b the element is accessed in a blockwise fashion from right to left. The lower bound (lb) of the loop is the iteration that is executed first. If the loop has negative stride, the upper bound (ub) is lower than the lower bound. In our method, the parts of an array given to each worker are pairwise disjoint.

Both figures also show the `offset` and `blocksize` given in the annotation. In Figure 4a the `offset` denotes the number of elements at the beginning of the array that are never accessed. In this case the `blocksize` has a positive value, since its added a number of times to the `offset` to obtain the index of the block. In Figure 4b, the `offset` is the index of the first element to be accessed in the first loop iteration. The `blocksize` is negative in this case since it is subtracted a number of times from the `offset`.

The following examples show how data should be annotated in different kinds of loops. The array `a` in these examples are arrays of `100` elements.

Example 1
We start with a simple example:

```
/*dist
  hosts = 2, threads = 4,
  a [0, 1] block mod_inout
*/
for(int i = 0; i < 100; i++)
    a[i] = i;
```

Each iteration uses only one element from the array, with the first iteration using the element 0 and every next iteration using the next element.

Example 2
The next example shows how to use the `offset`*:*

```
/*dist
  hosts = 2, threads = 4,
  a [5, 1] block mod_inout
*/
for(int i = 5; i < 65; i++)
    a[i] = i;
```

In this example, element `5` *from the array is the element accessed in the first iteration so the* `offset` *has to be* `5`*. Each iteration still accesses only one element. Also, in this example, only elements* `5` *to* `65` *are distributed.*

Example 3
In the next example, every iteration accesses `10` *elements:*

```
/*dist
  hosts = 2, threads = 4,
  a [50, 10] block mod_inout
*/
for(int i = 5; i < 10; i++)
    for(int j = 0; j < 10; j++)
        a[10 * i + j] = i;
```

Note that in this example the outer loop is being parallelized. The `blocksize` *is set to* `10`*, since the inner loop accesses* `10` *elements. The first element of the first block accessed is element* `50`*, so* `offset` *is* `50`*.*

Example 4
The next example is the same as the previous one, except for the different parallel loop stride:

```
/*dist
  hosts = 2, threads = 4,
  a [50, 10] block mod_inout
*/
for(int i = 5; i < 10; i += 2)
    for(int j = 0; j < 10; j++)
        a[10 * i + j] = i;
```

Changing the loop stride makes no difference in the annotation, however, since every iteration still accesses `10` *elements. In this example, each workers receives* `20` *elements, because the stride is now* `2`*. Only half of the elements are used, however.*

Example 5
The next example has a negative stride:

```
/*dist
  hosts = 2, threads = 4,
  a [99, -1] block mod_inout
*/
for(int i = 99; i >= 0; i--)
    a[i] = i;
```

Since changing the direction of the loop causes the elements of the array to be accessed in a different direction, we have to annotate these loops differently as explained previously. The first array element accessed is element `99`*, so the offset is annotated to be* `99`*. Every iteration accesses only one element, but because of the changed direction the* `blocksize` *is now* `-1`*.*

Example 6
This example is a slight modification of Example 4:

```
/*dist
  hosts = 2, threads = 4,
  a [99, -10] block mod_inout
*/
for(int i = 9; i >= 5; i -= 2)
    for(int j = 0; j < 10; j++)
        a[10 * i + j] = i;
```

Again, because of the negative stride, the annotations have changed accordingly.

4 Implementation overview

An overview of our method for exploiting loop parallelism is given in Figure 5. In this figure, the lines indicate remote method invocations from the client to the loop server. In addition, the arrows indicate data flow between client and loop server.

We have one application server running on every host that is involved in the parallel execution of an application. These hosts are also specified in a file called `".hostlist"`, which is used by the client to obtain the location of the loop servers. For each parallel loop, a loop server is generated. These loop servers are implemented as remote objects, which are created and registered by the application server. Each loop server maintains a number of worker threads, which actually perform the loop iterations. The loop servers implement methods for control and data transfer between the application and the loop servers, which are called from within the restructured loop.

The restructured loop first sends the data global to all workers to the server by invoking `setShared()` on the loop server. Next, the individual workers are given their specific data and started with by invoking `startJob()` on the loop server. When all workers are started, the application waits for each worker in turn to finish by invoking `waitJob()` after which the application collects the data to be returned by invoking the corresponding `get`-method for every block of data. Finally the job is removed from the loop server by invoking `deleteJob()` on it. Since multiple blocks of data may need to be returned to the client, the job cannot be removed within the invocation of any `get`-method.

At the client side, remote references to the loop servers are obtained in the constructor for the object containing the method with the parallel loop. To make sure that the servers are contacted, the lookup code is placed in every constructor. Contacting the servers in the constructors is efficient in case the loop is executed multiple times during the life time of the containing object. If the parallel loop is placed in a static method, the lookup procedure is performed by the static initializer for the class.

The application server and the loop server are generated by the compiler, which also transforms the parallel loop. Each loop server extends the `DistributeServer` class from the package `distribute`, which provides methods to maintain and synchronize with workers. The `distribute` package also contains the code that establishes connections to the loop servers for the clients.

In the next section, we will show how the `distribute` package is implemented. After that we will see how the application specific code for clients and servers is implemented in Section 6.

5 The `distribute` package

The `distribute` package contains classes which are used by every specific application that is parallelized.

5.1 The `DeepCopy` interface

The `DeepCopy` interface makes sure that a real (deep) copy of an object can be generated. As shown in Figure 6 it contains one method called `deepCopy()`, which the user needs to supply for every globally accessible object annotated `mod_in`. This method should make an exact copy of the object on which it is called, for instance by invoking the `clone()` method from the `Object` class. Note that this method cannot be called from a worker directly, since it is a protected method.

5.2 The `DistributeServerIntf` interface

The `DistributeServerIntf` is shown in Figure 7. The `DistributeServer` implements this interface, while the `lookup()` method from the `Distribution` class uses it as the type for

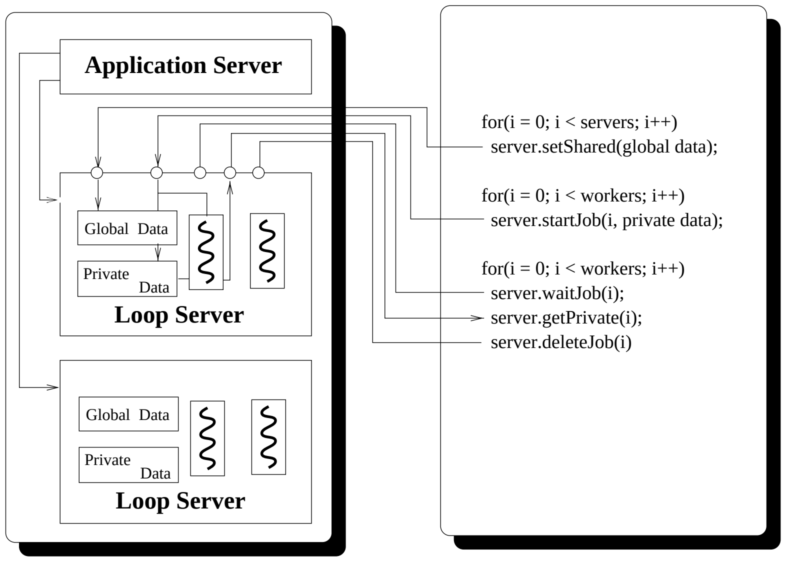

Application Server
Global Data
Private Data
Loop Server
Global Data
Private Data
Loop Server
for(i = 0; i < servers; i++)
server.setShared(global data);
for(i = 0; i < workers; i++)
server.startJob(i, private data);
for(i = 0; i < workers; i++)
server.waitJob(i);
server.getPrivate(i);
server.deleteJob(i)

Figure 5: Parallelization overview

```
public interface DeepCopy
{
  public Object deepCopy();
}
```

Figure 6: The `DeepCopy` interface

```
public interface DistributeServerIntf extends Remote { }
```

Figure 7: The `DistributeServerIntf` interface

```
public class Distribution
{
  private static final int PORT = 1115;
  private static Vector    hosts = new Vector();

  public static void lookup(DistributeServerIntf[] ds, String name)
  {
    try {
      for(int i = 0; i < ds.length; i++) {
        ds[i] = (DistributeServerIntf)
                      Naming.lookup("//" + (String)hosts.elementAt(i) + ":" + PORT + "/" + name);
        System.out.println(
                     "Connection to " + (String)hosts.elementAt(i) + ":" + PORT + " established");
      }
    }
    catch(Exception e) {
      e.printStackTrace();
      System.exit(1);
    }
  }

  static
  {
    try {
      String host;
      BufferedReader dis =
                     new BufferedReader(new InputStreamReader(new FileInputStream(".hostlist")));
      while((host = dis.readLine()) != null)
        hosts.addElement(host);
      dis.close();
    }
    catch(Exception e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

Figure 8: The `Distribution` class

generic remote objects created for the parallel loops (see Figure 8).

5.3 The `Distribution` class

The `Distribution` class takes care of setting up connections to the servers at the client side. The static initializer reads a list of hostnames from a file `".hostlist"`. The `lookup()` method is given an array of uninitialized remote references of a certain length. It initializes these references by looking up up the remote object with the give `name` at registry on the hosts obtained from the hostlist. If the length of the array of remote references is `n` than the remote object is looked up on the first `n` hosts from the hostlist. If not enough hosts are specified in the hostlist an error is generated. Specifying more hosts than needed is permitted, however. Its implementation is shown in Figure 8 and is straightforward.

5.4 The `DistributeServer` class

The `DistributeServer` class is the superclass of all remote objects created for a parallel loop. It provides methods for maintaining the workers that execute the iterations of the loop. The

```
public abstract class DistributeServer extends UnicastRemoteObject implements DistributeServerIntf
{
  protected final int           MAXTHREADS = 100;
  protected final int           IDLE = 0;
  protected final int           BUSY = 1;
  protected final int           READY = 2;

  protected int                 nrthreads;
  protected int                 curid;
  protected int[]               status;

  protected DistributeServer() throws RemoteException
  {
    nrthreads = 0;
    curid = 0;
    status = new int[MAXTHREADS];
    for(int i = 0; i < MAXTHREADS; i++)
      status[i] = IDLE;
  }

  public synchronized void jobFinished(int jid)
  {
    status[jid] = READY;
    notify();
  }

  protected synchronized int getFreeJobEntry();
  {
    if(nrthreads == MAXTHREADS)
      System.exit(0);

    while(status[curid] != IDLE)
      curid = (curid + 1) % MAXTHREADS;

    nrthreads++;
    status[curid] = BUSY;
    return curid;
  }

  public synchronized void waitJob(int jid);
  {
    try {
      while(status[jid] != READY)
        wait();
    }
    catch(InterruptedException e) { }
  }

  public void deleteJob(int jid);
  {
    nrthreads--;
    status[jid] = IDLE;
  }
}
```

Figure 9: Outline of the DistributeServer class

specific servers in turn are managed by a separate server, which has type `AppServer`, as we will see in Section 6.

An outline of the `DistributeServer` class is shown in Figure 9. The constructor initializes its table of workers. The part of the table implemented in this class consists of only the status for each worker. The specific servers, in addition, store the data that is to be returned to the client in a similar way as we will see in Section 6. The `jobFinished()` method is called by the workers to indicate to the server that their job has completed. If the server was waiting for this worker, it is signaled.

The `getFreeJobEntry()` method returns a free job entry from the server table. If no free entries are available the server exits. There is no possibility to wait for an entry to become free, because the results to be copied back are also stored in the table. These results, however, are not collected by the client before all jobs have been started, so waiting for an entry to become free would lead to a deadlock. Note that the number of threads are bounded by annotations, so if this annotation does not specify a larger number of threads than MAXTHREADS, overflow could never occur.

The `waitJob()` method waits for the specified job to finish. The signal that awakes the server comes from the `jobFinished()` method, which is invoked by a worker upon its termination. The `deleteJob()` method removes the given job from the server table.

6 Source Transformations

This section describes the actual transformations performed by the compiler to exploit parallelism. As described in our overview, the resulting source code always includes the following parts:

- The original code, in which the parallel loops have been replaced by invocations to remote loop servers.
- For every parallel loop, a remote interface is created, which contains the methods that can be invoked on it remotely as described in Section 4.
- for every parallel loop, a remote object is created, which implements the corresponding remote interface.
- An application server is created.

Each of these parts will be described in the following subsections. We will describe the code generation and transformation using the example given in Figure 10. For clarity, the loop body is left empty, but assume that the five private variables are actually used in the loop body. These five variables each correspond to a different class of variables as described in Section 3.2. We annotate these accordingly, using a simple data distribution. Sections 6.4 through 6.3 describe the resulting code produced by the compiler.

6.1 The loop server remote interface

The remote interface for the server that handles the parallel loop is shown in Figure 11. Its name is constructed from the loop and method in which the parallel loop was located. Since a method can have multiple parallel loops an additional unique number is also in the name.

```
public class Loop
{
  private GlobalUnmod       sunm;
  private GlobalModIn       smin;
  private PrivateUnmod[]    punm = new PrivateUnmod[10];
  private PrivateModIn[]    pmin = new PrivateModIn[10];
  private PrivateModInout[] pmio = new PrivateModInout[10];
  ...
  public Loop()
  ...
  public void start()
  {
    ...
    /*dist block,
      hosts = 2, threads = 4,
      sunm unmod, smin mod_in,
      punm [0, 1] block unmod,
      pmin [0, 1] block mod_in,
      pmio [0, 1] block mod_inout
    */
    for(int i = 0; i < 10; i++) {
      ... = sunm...
      smin = ...;
      ... = punm[...];
      pmin[...] = ...;
      pmio[...] = ...;
    }
    ...
  }

  public static void main(String[] Args)
  {
    Loop l = new Loop();
    l.start();
  }
}
```

Figure 10: Example application

```
public interface Loopstart0Intf extends distribute.DistributeServerIntf

  public void setShared(GlobalUnmod sunm, GlobalModIn smin)
                                                          throws java.rmi.RemoteException;
  public int startJob(int workerId, int low, int high, int stride,
                    PrivateUnmod[] punm, PrivateModIn[] pmin, PrivateModInout[] pmio)
                                                          throws java.rmi.RemoteException;
  public void waitJob(int jobid) throws java.rmi.RemoteException;
  public PrivateModInout[] getpmio(int jobid) throws java.rmi.RemoteException;
  public void deleteJob(int jobid) throws java.rmi.RemoteException;
```

Figure 11: The loop server remote interface

Every loop server remote interface contains the following methods:

- `setShared()`, which is supplied with all variables global to all workers. It transfers the supplied data the specified server.
- `startJob()`, which is supplied a `workerId` together with the set of iterations to be performed by the worker and all private data for the worker. This method transfers the data and starts a worker.
- `waitJob()`, which is given the job identification number to wait for. It blocks until the worker notifies the server of its completion.
- For every piece of data that is to be returned to the client a `get`-method is present, which transfers the result value of the given job number to the client.
- `deleteJob()` removes the data belonging to the given job from the server.

6.2 The loop server

The implementation of the loop server is shown in Figure 12. The server stores a copy of all global data and a copy of private data to be returned to the client for each worker. Managing the workers is done for the most part in the `DistributeServer` class. The specific servers are primarily concerned with storing the data and passing data to the servers. The `startJob()` method first obtains a free table entry and stores a copy of the job private data in that entry. The job is than started, after which the job identification number `jobId` is returned to the client.

The worker code is present in the same file. The implementation of a worker is shown in Figure 15. The main reason for storing a local copy of a reference to all relevant data is to speed up access to these data. Figure 13 shows how different types of data are accessed. The constructor and the first part of the `run()` method are primarily concerned with initializing the local copies.

That part of the original private data that is needed by a worker is mapped into an an array which has just the size required to hold that data. Since all array lower bounds in Java are `0` the actual indices calculated in the loop body are no longer valid in the worker. The skew values, calculated prior to loop execution by the worker, compensates for this change in indices. This effect is shown in Figure 14. Basically, the skew is the index in the original array of the element

```
public class Loopstart0 extends distribute.DistributeServer implements Loopstart0Intf
{
  GlobalUnmod sunm;
  GlobalModIn smin;
  PrivateModInout[][] pmio;

  public Loopstart0() throws java.rmi.RemoteException
  {
      pmio = new PrivateModInout[MAXTHREADS][];
  }

  public void setShared(GlobalUnmod sunm, GlobalModIn smin) throws java.rmi.RemoteException
  {
    this.sunm = sunm;
    this.smin = smin;
  }

  public int startJob(int workerId, int low, int high, int stride,
                      PrivateModIn[] pmin, PrivateModInout[] pmio) throws java.rmi.RemoteException
  {
    int jobId = getFreeJobEntry();
    this.pmio[jobId] = pmio;
    Loopstart0Worker aWorker =
                             new Loopstart0Worker(jobId, this, workerId, low, high, stride, pmin);
    aWorker.start();
    return jobId;
  }

  public PrivateModInout[] getpmio(int jobid)
  {
    return this.pmio[jobid];
  }
}
```

Figure 12: The loop server

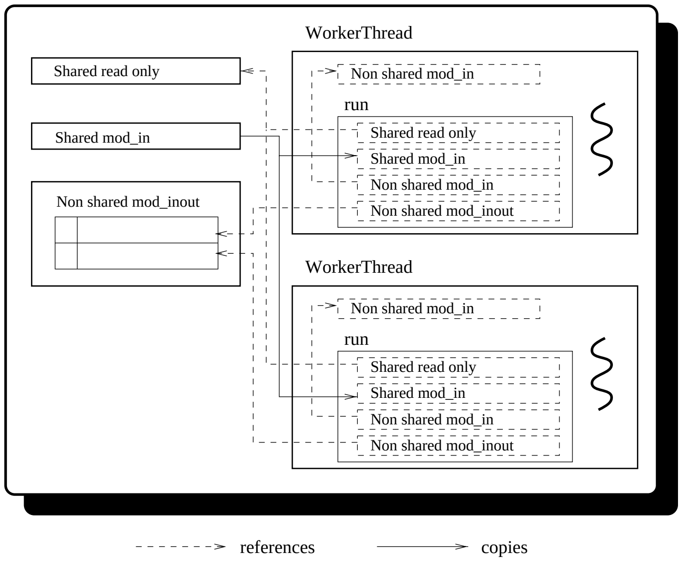


Figure 13: Overview of the data storage at the server

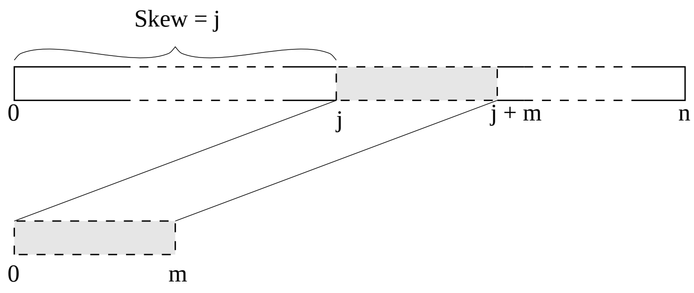


Figure 14: Array index skew

stored at index `0` at the worker. In the loop body at the worker, every index `i` into a worker private array `a` needs to be replaced with (`i - aSkew`).

After the loop has been executed, the server is notified of job completion by invocation of `jobFinished()` on the server by the worker.

6.3 The application server

The implementation of the application server is shown in Figure 16. Except for the registering of remote loop servers by invoking `rebind()`, the application server looks the same for every application. The application server installs a security manager and its own registry, after which it creates every loop server and registers these.

6.4 Parallelizing the original code

An outline of the new client code is given in Figure 17. The client code first has to set up connections to every server, which is done in the constructor by invoking the `lookup()` method from the `Distribution` class. The code will actually be placed in every constructor for the object, which is why a check is made to see whether the lookup has already been performed.

Figure 18 shows the result of restructuring the original parallel loop. Prior to restructuring the parallel loop, the compiler performs loop normalization, which yields a loop with a lower bound of `0` and a unit stride. The upper bound in this example is given by the expression (`10 - 0 - 1 + 1`). Loop normalization is described extensively in literature, see for instance [11]. After having performed loop normalization the index variable in the loop body is replaced accordingly.

The variable `blockIters` holds the number of loop iterations to be executed by each worker. This number is calculated by dividing the upper bound of the normalized loop by the number of workers. Some workers may actually perform less iterations because of rounding off this number and because the total number of loop iterations need not be a multiple of the number of workers. For each worker private variable a new array is constructed, which holds the portion of the original array that a worker needs.

The first loop multicasts the global data to all workers. In the second loop, the worker private arrays are divided, after which each worker is started, supplied with their data and loop iterations. Note that the second loop contains two separate copies of the code that performs the array partitioning, because each array may be distributed differently. The structure of these two

```
class Loopstart0Worker extends Thread
{
  private int jobId;
  private Loopstart0 ds;
  private int workerId;
  private int low, high, stride;
  PrivateModIn[] pmin;

  Loopstart0Worker(int jobId, Loopstart0 ds, int workerId, int low, int high,
                                                            int stride, PrivateModIn[] pmin)
  {
    this.jobId = jobId;
    this.ds = ds;
    this.workerId = workerId;
    this.low = low;
    this.high = high;
    this.stride = stride;
    this.pmin = pmin;
  }

  public void run()
  {
    int jobId = this.jobId;
    int low = this.low;
    int high = this.high;
    int stride = this.stride;
    int workerId = this.workerId;
    Loopstart0 ds = this.ds;
    GlobalUnmod sunm = ((Loopstart0)ds).sunm;
    GlobalModIn smin = (GlobalModIn)(((Loopstart0)ds).smin.clone());
    PrivateUnmod[] punm = this.punm;
    PrivateModIn[] pmin = this.pmin;
    PrivateModInout[] pmio = ((Loopstart0)ds).pmio[jobId];

    int punmSkew;
    if(1 < 0)      punmSkew = 1 + 0 + (Math.abs(1) * 1 * high);
    else           punmSkew = 0 + (Math.abs(1 ) * 1 * low);
    int pminSkew;
    if(1 < 0)      pminSkew = 1 + 0 + (Math.abs(1) * 1 * high);
    else           pminSkew = 0 + (Math.abs(1 ) * 1 * low);
    int pmioSkew;
    if(1 < 0)      pmioSkew = 1 + 0 + (Math.abs(1) * 1 * high);
    else           pmioSkew = 0 + (Math.abs(1 ) * 1 * low);

    for(int i = low; i < high; i += stride)
      { }

    ds.jobFinished(jobId);
  }
}
```

Figure 15: The implementation of a worker

```
public class AppServer
{
  private static final int PORT = 1115;
  private static String hostname;

  public static void main(String[] Args)
  {
    System.setSecurityManager(new java.rmi.RMISecurityManager());
    try {
      hostname  = java.net.InetAddress.getLocalHost().getHostName();
      java.rmi.registry.LocateRegistry.createRegistry(PORT);
      java.rmi.Naming.rebind("//" + hostname + ":" + PORT + "/Loopstart0", new Loopstart0());
      System.out.println("Server ready.");
    }
    catch(Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
  }
}
```

Figure 16: The application server

```
public class Loop {
  private Loopstart0Intf[] Loopstart0 = null;
  private GlobalUnmod sunm;
  private GlobalModIn smin;
  private PrivateUnmod[] punm = new PrivateUnmod[10];
  private PrivateModIn[] pmin = new PrivateModIn[10];
  private PrivateModInout[] pmio = new PrivateModInout[10];

  Loop()
  {
    if (Loopstart0 == null) {
      Loopstart0 = new Loopstart0Intf[2];
      distribute.Distribution.lookup(Loopstart0, "Loopstart0");
    }
  }

  public void start() { ... }

  public static void main(String[] Args)
  {
    Loop l = new Loop();
    l.start();
  }
}
```

Figure 17: An outline of the transformed client code

```
public void start()
{
  try {
    int blockIters = ((int) Math.ceil(((double) ((((10 - 0) - 1) + 1) / 1)) / ((double) 8)));
    int[] jobId = (new int[8]);
    PrivateModIn[] pminBlock = (new PrivateModIn[blockIters * (Math.abs(1) * Math.abs(1))]);
    PrivateModInout[] pmioBlock = (new PrivateModInout[blockIters * (Math.abs(1) * Math.abs(1))]);
    int curItem = 0, curJlb = 0, curJub = 0, curJst = 0, curKlb = 0, curKub = 0, curKst = 0;

    for(int counterI = 0; counterI < 2; counterI++)                    /* Distribute global data */
      Loopstart0[counterI].setShared(sunm, smin);

    for(int counterI = 0; counterI < 8; counterI++) {
      curJlb = (counterI * blockIters);                         /* calculate subset of iterations */
      curJub = Math.min(curJlb + blockIters, (((10 - 0) - 1) + 1) / 1);
      curJst = 1;
      curItem = 0;                                                        /* Partition pmin array */
      if(1 < 0) {
        curKlb = Math.max((1 + 0) + (Math.abs(1) * (curJub * 1)), 0);
        curKub = ((1 + 0) + (Math.abs(1) * (curJlb * 1)));
      } else {
        curKlb = (0 + (Math.abs(1) * (curJlb * 1)));
        curKub = Math.min(0 + (Math.abs(1) * (curJub * 1)), 10 * Math.abs(1));
      }
      for(int counterK = curKlb; counterK < curKub; counterK++)
        pminBlock[curItem++] = pmin[counterK];
      curItem = 0;                                                        /* partition pmio array */
      if(1 < 0) {
        curKlb = Math.max((1 + 0) + (Math.abs(1) * (curJub * 1)), 0);
        curKub = ((1 + 0) + (Math.abs(1) * (curJlb * 1)));
      } else {
        curKlb = (0 + (Math.abs(1) * (curJlb * 1)));
        curKub = Math.min(0 + (Math.abs(1) * (curJub * 1)), 10 * Math.abs(1));
      }
      for(int counterK = curKlb; counterK < curKub; counterK++)
        pmioBlock[curItem++] = pmio[counterK];
      jobId[counterI] = Loopstart0[counterI % 2].startJob(counterI, curJlb, curJub, curJst,
                                                         pminBlock, pmioBlock); /* start worker */
    }

    for (int counterI = 0; counterI < 8; counterI++) {
      curJlb = (counterI * blockIters); /* recalculate subset of iterations */
      curJub = Math.min(curJlb + blockIters, (((10 - 0) - 1) + 1) / 1);
      curJst = 1;
      Loopstart0[counterI % 2].waitJob(jobId[counterI]);               /* Wait for job to finish */
      pmioBlock = Loopstart0[counterI % 2].getpmio(jobId[counterI]);          /* Get pmio result */
      curItem = 0;                                               /* Copy result in original array */
      if(1 < 0) {
        curKlb = Math.max((1 + 0) + (Math.abs(1) * (curJub * 1)), 0);
        curKub = ((1 + 0) + (Math.abs(1) * (curJlb * 1)));
      } else {
        curKlb = (0 + (Math.abs(1) * (curJlb * 1)));
        curKub = Math.min(0 + (Math.abs(1) * (curJub * 1)), 10 * Math.abs(1));
      }
      for(int counterK = curKlb; counterK < curKub; counterK++)
        pmio[counterK] = pmioBlock[curItem++];
      Loopstart0[counterI % 2].deleteJob(jobId[counterI]);             /* Remove job from worker */
      }
    }
  } catch(java.rmi.RemoteException e) { System.exit(1); }
}
```

Figure 18: Result of restructuring the parallel loop

```
/*dist block,
  hosts = 2, threads = 4,
  b unmod, matsize unmod,
  a [0, 1] block mod_in,
  c [0, 1] block mod_out
*/
for(int i = 0; i < matsize; i++)
  for(int j = 0; j < matsize; j++) {
    c[i][j] = 0;
    for(int k = 0; k < matsize; k++)
      c[i][j] += a[i][k] * b[k][j];
  }
```

Figure 19: Parallel matrix multiplication

pieces of code are similar, however, as is the code that copies the results from the server in the original array. The lower (`curJlb`) and upper bound (`curJub`) for a worker are calculated from the number of iterations for each worker. Note that the normalized loop is divided rather than the original loop.

Dividing the worker private arrays is dependent upon the order in which the loop iterations access the blocks. If these blocks are accessed from right to left, the then part of the `if`-statements is executed. The first and last (exclusive) element are than calculated as follows:

$$first = \max(1 + offset + |stride| * curJub * blocksize, 0)$$
$$last = 1 + offset + |stride| * curJlb * blocksize$$

In these formulae, the `offset` and `blocksize` are as annotated and the `stride` is the stride of the original parallel loop. If the original array is accessed from left to right, the formulae are as follows (where upper bound is the upper bound of the original loop):

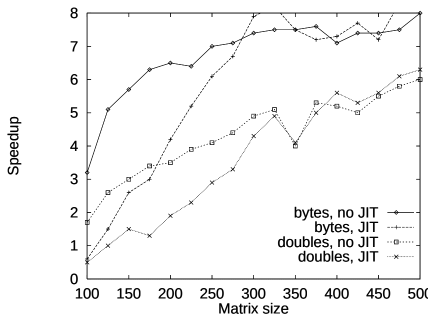
$$first = offset + |stride| * curJlb * blocksize$$
$$last = \min(offset + |stride| * curJub * blocksize, upperbound * |blocksize|)$$

The elements from the original array between `first` and `last` are copied in a separate copy, after which the workers are started. Each worker is supplied with its part of the array data. When all workers are started, the client waits for the workers to finish in the same order as they are started. If a worker is finished, the required data is retrieved by the client, which merges all subparts together in a similar way as it divided the original data, after which it removes the job from the server.

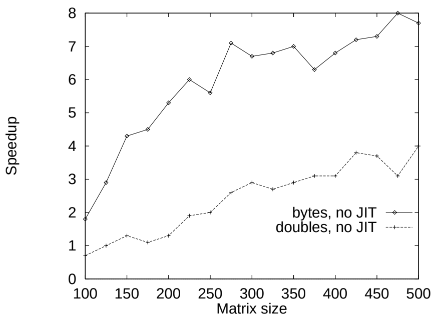
7 Performance

To test the performance of our parallelization method, we have parallelized a matrix multiplication algorithm. The parallel loop and its annotations are shown in Figure 19. Note that we parallelize the outer loop.

The two hosts that we were using in this project were two IBM RS/6000 G30 machines, each containing four PowerPC 604 processors running at 112 MHz and 256 Mbytes of RAM. Both



(a) Two multiprocessors



(b) Cluster of SPARCclassics

Figure 20: Matrix multiplication results

machines run the AIX 4.2 operating system, which supports true multithreading. The tests were compiled and run with the AIX version of the JDK version 1.1.2. The machines were connected by an Ethernet 10Mbit/s network.

The sequential version was run without performing our transformations, instead of using one client and one server. The parallel versions had the client running as a separate process on one of the two machines. We have run our test using both byte and double matrices and both with and without JIT enabled. The results are shown in Figure 20a.

As the graph shows, both with and without JIT, a speedup of eight can be obtained when multiplying byte matrices. The superlineair speedup for double matrices with JIT disabled is caused by an external event on the machine. Without JIT a higher speedup can be obtained for smaller matrices. This is because the ratio of computing time and transfer time is higher when JIT is disabled, since JIT only affects computation speed. For double matrices the speedup is lower than for byte matrices. This can be attributed to the high overhead of object serialization in Java. Again, for small matrices, the speedup obtained without JIT is better than the speedup obtained with JIT. We have also tested matrix multiplication over an ATM switching network. The amount of data, however, was too small to benefit from a high speed connection. The communication overhead was still dominated by object serialization.

The same experiments were run on a cluster of SPARCclassic stations running at 50MHz and containing 48 Mbytes of memory. The operating system used is Solaris version 5.5. The experiments were done with JDK 1.1.3. Again, the client ran as a separate process on one of the machines running a server. The servers each contained one worker, since no true multithreading exists on these machines. The results of these tests are shown in Figure 20b. At the time of testing, JIT was not installed on these machines.

The speedups obtained when using byte matrices are still good. For large enough arrays, a speedup of eight can be obtained. When using double matrices, however, the obtained speedup is a significantly lower. This is due to the lower processor speed, which makes serialization of doubles even more expensive and thus deteriorates performance.

To assess whether our method can be used to parallelize other applications, we have tested our method on the Image/J application [8]. Image/J is a prototype image processing program, which implements some well know image processing algorithms like smoothing.

The method containing the loop is shown in Figure 21. The left side shows the original code, the right side shows the code somewhat modified with our annotations added.

Some minor changes had to be made to the code in order to be able to parallelize it. First, the drawing of the progress bar was removed; it is not possible for a server to produce any output, since this output will not show up on the client machine. The progress bar indicated how much of the job is completed, which is of no use in a parallel version.

Furthermore, to remove output dependences on local variables declared outside the loop, we declared these inside the loop. Since they were not carrying dependences between different iterations of the y-loop, this preserves the original semantics.

We cannot distribute the `pixels2` array row wise, since multiple rows of `pixels2` are accessed to compute one row of `pixels`. Distributing `pixels2` would lead to one worker needing information that is only stored on another worker, which is not supported in our method. The `pixels2` array, however, can be distributed. Analysis of the access pattern, reveals an offset and blocksize for `pixels`.

Since our experiments with matrix multiplication showed that the best results were obtained with byte matrices, we only use a gray scale picture for our experiment (stored in a byte array in Image/J). We compare the performance of our loop on a different number of workers on our two multiprocessors and on our cluster of SPARCclassics. On the multiprocessor machines, the workers were equally divided among the two machines, while a different number of hosts were used on the cluster of SPARCclassics, where each server contained one worker. To obtain a fair comparison, we measured the serial time after having modified the program as described above. The sequential version (workers = 1) was run without applying our loop transformation.

The speedups obtained in this test are shown in Figure 22. The main reason for the low performance is the large amount of data that has to be transferred to every host, due to the fact that the source image could not be partitioned, while the amount of computation for each element is relatively small.

8 Conclusions

In this document, a method of exploiting implicit loop parallelism using multiple multithreaded servers is proposed. The code to execute the iterations in parallel can be generated by a Java restructuring compiler. We have implemented our transformations as a module in `Javar` [3].

Our approach is to divide the loop iterations among a number of workers, which run as a separate thread in a server process. To make this work, we also sent the required data to the workers. Data that is used by every worker is sent to each server once. For arrays, a simple distribution method has been implemented to minimize communication overhead. When the original application has transferred all data and started every worker, it waits for each worker to finish. If a worker is finished, the application transfers the data back and removes the workers from the server.

Detection of parallel loops, the number of workers to use and the data distribution method are all guided by annotations in the original source code in the form of comments ignored by other compilers.

```
void medianFilter() {
  byte[] pixels2;
  int x, y, offset, rowOffset;
  int p1, p2, p3, p4, p5, p6, p7, p8, p9;

  pixels2 = new byte[width * height];
  System.arraycopy(pixels,0,pixels2,0,width*height);
  rowOffset = width;
  int[] values = new int[10];







  for (y = yMin; y <= yMax; y++) {

    offset = xMin + y * rowOffset + 1;
    p2 = (pixels2[offset - width - 2] & 0xff);
    p3 = (pixels2[offset - width - 1] & 0xff);
    p5 = (pixels2[offset] & 0xff - 2);
    p6 = (pixels2[offset - 1] & 0xff);
    p8 = (pixels2[offset + width - 2] & 0xff);
    p9 = (pixels2[offset + width - 1] & 0xff);
    offset = xMin + y * rowOffset;
    for (x = xMin; x <= xMax; x++) {
      p1 = p2;
      p2 = p3;
      p3 = (pixels2[offset - rowOffset + 1] & 0xff);
      p4 = p5;
      p5 = p6;
      p6 = (pixels2[offset + 1] & 0xff);
      p7 = p8;
      p8 = p9;
      p9 = (pixels2[offset + rowOffset + 1] & 0xff);
      values[1] = p1;
      values[2] = p2;
      values[3] = p3;
      values[4] = p4;
      values[5] = p5;
      values[6] = p6;
      values[7] = p7;
      values[8] = p8;
      values[9] = p9;
      pixels[offset++] =
                  (byte)(findMedian(values) & 0xff);
    }
    if (y%20==0)
      showProgress((double)(y-roiY)/roiHeight);
  }
  hideProgress();
}
```

```
void medianFilter() {
  byte[] pixels2;
  int rowOffset;
  int low__ = yMin, up__ = yMax;

  pixels2 = new byte[width * height];
  System.arraycopy(pixels,0,pixels2,0,width*height);
  rowOffset = width;
  /*dist
    hosts = 2, threads = 4,
    xMin unmod, xMax unmod, rowOffset unmod,
    width unmod, pixels2 unmod, low__ unmod,
    pixels [xMin + (low__ * rowOffset), rowOffset]
                                block mod_inout
  */
  for (int y = low__; y <= up__; y++) {
    int[] save = new int[10];
    int offset = xMin + y * rowOffset + 1;
    int p1, p2 = (pixels2[offset - width - 2] & 0xff);
    int p3 = (pixels2[offset - width - 1] & 0xff);
    int p4, p5 = (pixels2[offset] & 0xff - 2);
    int p6 = (pixels2[offset - 1] & 0xff);
    int p7, p8 = (pixels2[offset + width - 2] & 0xff);
    int p9 = (pixels2[offset + width - 1] & 0xff);
    offset = xMin + y * rowOffset;
    for (int x = xMin; x <= xMax; x++) {
      p1 = p2;
      p2 = p3;
      p3 = (pixels2[offset - rowOffset + 1] & 0xff);
      p4 = p5;
      p5 = p6;
      p6 = (pixels2[offset + 1] & 0xff);
      p7 = p8;
      p8 = p9;
      p9 = (pixels2[offset + rowOffset + 1] & 0xff);
      save[1] = p1;
      save[2] = p2;
      save[3] = p3;
      save[4] = p4;
      save[5] = p5;
      save[6] = p6;
      save[7] = p7;
      save[8] = p8;
      save[9] = p9;
      pixels[offset++] =
                  (byte)(findMedian(save) & 0xff);
    }


  }

}
```

Figure 21: Loop parallelization in Image/J

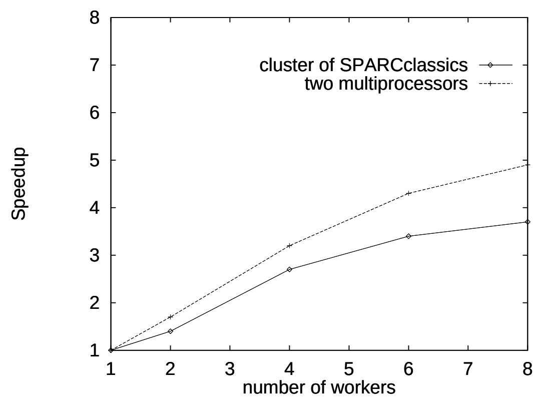


Figure 22: Performance of `medianFilter()`

We have shown that applications may show good speedup when the transformations described are applied. Our matrix multiplication example for byte matrices obtains a speedup of about seven to eight when run on two multiprocessor machines with four processors each. The same speedup can be obtained when using a cluster of eight workstations. When using more complicated data, the performance degrades, due to Java's object serialization method.

To show that our method can be used to parallelize real applications, we have parallelized some loops in the **Image/J** package. This experience indicated that finding the right data distribution annotation requires some analysis of the program. Parallelizing programs with our tool, however, is still a great deal simpler than performing the transformations by hand.

We have implemented only a simple scheduling and data distribution method. In the future, more sophisticated scheduling and data distribution methods could be added. To minimize overhead for finer grained loops, multiversion code could be generated, that would execute the loop serially if the number of loop iterations is small.

References

[1] U. Banerjee. *Dependence Analysis.* A book Series on Loop Transformations for Restructuring Compilers. Kluwer, Boston, 1997.

[2] A.J.C. Bik and D.B. Gannon. Automatically Exploiting Implicit Parallelism in Java. *Concurrency, Practice and Experience*, 9(6), 1997.

[3] A.J.C. Bik, J.E. Villacis, and D.B. Gannon. `JAVAR` *manual.* Computer Science Department, Indiana University, 1997. This manual and the complete source of `javar` is made available at `http://www.extreme.indiana.edu/hpjava/`.

[4] H.M. Deitel and P.J. Deitel. *Java How to Program.* Prentice Hall, Upper Sadle River, New York, 1997.

[5] J. Flanagan. *Java in a Nutshell.* O' Reilly, Sebastopol, California, second edition, 1997.

[6] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley Developers Press, 1996.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison Wesley, Massachusetts, 1996.

[8] W. Rasband. *Image/J version 0.46.* National Institutes of Health, USA, 1997. Package available at http://rsb.info.nih.gov/IJ/.

[9] K.S. Siyan and J.L. Weaver. *Inside Java.* New Riders Publishing, Indianapolis, Indiana, 1997.

[10] Sun Microsystems. *Java Remote Method Invocation Specification*, feb 1997.

[11] M.J. Wolfe. *High Performance Compilers for Parallel Computers.* Addison-Wesley, Redwood City, California, 1996.