

JOB SIZE FOR INTERNET PARALLEL COMPUTING

PEIYI TANG

Department of Mathematics and Computing
University of Southern Queensland
Toowoomba QLD 4350 Australia

JINGLING XUE

School of Mathematical and Computer Sciences
University of New England
Armidale NSW 2351 Australia

Abstract

In this paper, we present a performance model and the experimental results of parallel execution of Java Remote Method Invocation (RMI). The Java RMI is used for remote execution of parallel jobs on the Internet. The main use of the model is to determine the job size of the distributed objects that are suitable for the meta-computing on the Internet.

1 INTRODUCTION

The advent of mobile programming languages [1] such as Java [2] opens the entire Internet for parallel computing. While the parallel computing on clusters of workstations (COWs) also uses general network protocols such as TCP/IP for interprocessor communication, there is a fundamental difference: the processors of a COW share the same (networked) file system, but the computers on the Internet don't. The parallel computing on the Internet is also known as *meta-computing*. The idea of meta-computing is to explore the huge aggregate computing power of millions of computers on the Internet for large computational problems. With an appropriate economic model to reward contribution, the computer owners may be willing to sell the otherwise idle CPU cycles or exchange them for other computing resources such as software licenses [3].

Meta-computing will not attract many contributors without mobile programming languages. Mobile programming languages provide mobile code which is the "software that travels on a heterogeneous network, crossing protection domains, and automatically executed upon arrival at the destination" [1]. Early projects on the meta-computing such as Nimrod [4] do not use mobile code. Without mobile code, the program for the remote host needs to be transferred to its file system in advance. This means that the program has to be regarded as trusted code on the remote host. Another obstacle is the hassle to transfer the

code manually, for instance, using ftp.

Mobile code eliminates all these obstacles. With security measures for the mobile code such as Java byte-code [5], the owner of the remote host can have a peace of mind that *only* CPU cycles of his(her) machine are used. No prior code transfer is needed.

There has been a strong research interest in the meta-computing based on mobile code in the last few years [6, 7, 8, 9, 10]. Some of the projects have successfully run large applications showing reasonable speedup for certain applications. It is well known that the Internet has large communication latency and not all types of applications are suitable for meta-computing. It is believed that only those applications which can be decomposed to independent tasks of substantial sizes are possible candidates. The question here is what sizes of the tasks are large enough for meta-computing. To answer this question, we need a performance model to guide the run-time system to make intelligent decisions on partitioning and decomposition of the applications.

Predicting the network latency and the remote execution time of tasks on the Internet is difficult. To achieve a balanced job distribution and high speedup, we probably have to rely on dynamic scheduling and load balancing. However, we still need to rule out the tasks that are too small at the first place. In this paper, we use the simplest network configuration, local network, and a simple task model to determine the threshold of job size. The compilers and run-time systems can rule out the task candidates whose sizes are below this threshold.

The paper is structured as follows. Section 2 presents a meta-computing model based on the concept of *mobile objects*. This model is based on the recent Java technologies [11]: Java RMI, Java Object Serialization and Java Core Reflection. In Section 3, we present our performance model for the threshold of job size for meta-computing. The result of the experiments and measurements are presented and analyzed in Section 4. Section 5 concludes the paper with a discussion of future work.

2 A META-COMPUTING MODEL

A basic requirement for meta-computing is the seamless movement of computational tasks (including both the code and the data) from one host to another. With the recent Java technologies, RMI and Object Serialization, this requirement can be implemented easily.

A remote method invocation can pass a serializable objects to the remote host. The remote host then runs the required computation upon the passed object. The code and data of the computation are encapsulated in this object: The data fields are passed by value through Object Serialization and the code is passed to the remote host through the normal Java class loading mechanism. This kind of objects are called *mobile objects*. The remote method invocation can also return the local host a serializable object as the result of the computation. The returned object can be the same mobile object after it is processed or any other object that encapsulates the result of the computation.

To enable the remote execution of any method of any mobile object without recompiling the code of the local host, we need a generic class, `Job`, to wrap up the name of the method, the mobile object and the arguments. The API of the `Job` class is defined as follows:

```
public class Job implements Serializable
{
    protected Object target; // the mobile object
    protected Object[] pars;
    String mtdName;
    // arguments and name of the method invoked
    protected Object rtn; // return object

    Job(Object obj, String methodName,
        Object[] args);

    public void execute()
        throws InvocationTargetException, ...
    // execute the method at the remote host

    public Object getObject();
    //return the processed mobile object
    public Object getReturn();
    //return the return object
}
```

The mobile object which encapsulates all its state and data is passed to the `Job` upon the construction and so are the name of the method and its arguments. The implementation of `execute()` uses Java Reflection to run the specified method of the mobile object on the remote host. `getObject()` returns the mobile object itself after its data and state are changed and processed. The specified method may have a return object. `getReturn()` is used to get this return object

of the method. All the mobile object, the arguments of the method and the return object must be serializable. The `Job` object also has to be serializable itself.

To enable the local host to pass a `Job` object to the remote host, the remote host can implement a remote method, say, `push(Job job, ...)`. Then, the local host can call this method to pass the `Job` object to be executed on the remote host. The remote method `push(Job job, ...)` can be implemented either in the synchronous or asynchronous fashions. The synchronous `push(Job job, ...)` does not return until the `Job` object has been processed and it returns the same `Job` object as illustrated by the following code:

```
public Job push(Job job, ...)
    throws InvocationTargetException, ...
{
    ...
    job.execute();
    return job;
}
```

The asynchronous `push(Job job, ...)` returns immediately. The remote host puts the `Job` object in its input queue. The local host has to implement a remote method, say, `done(Job job, ...)`, so that the remote host can call this method to pass the `Job` object back to the local host after it is processed.

The remote host starts with loading an applet from the local host. That applet and the master program of the local host can establish the connection by exchanging the references of the remote objects on both sides. After that, the local host can construct `Job` objects and call `push(Job job, ...)` to pass them to the remote hosts for execution. All the code required by the remote host can be automatically loaded from the local host by the class loader of the applet.

During its lifetime, a mobile object can be processed by its different methods at different stages, by wrapping it up in different `Job` objects in pipeline. These `Job` objects can be processed in different remote hosts.

3 A PERFORMANCE MODEL

A `Job` object described in the previous section represents a self-contained computation which does not require interaction with other objects. The purpose of meta-computing is to run large number of `Job` objects on remote hosts to achieve high speedup. This implies that the RMIs for the multiple `Job` objects need to be run in parallel. The parallel RMI can be implemented by concurrent threads in Java, each of which makes a `push(Job job, ...)` RMI to a remote host.

It is well known that the network latency on the Internet is large. The software layers for RMI and network protocols add additional overheads on the cost

of RMI. To amortize this large overhead, the computation time of the RMI on the remote host should be sufficiently large. The computation time required for speedup depends on the actual overhead of the parallel RMI. However, modeling and predicting the overhead of RMI on the Internet is very difficult, if not impossible. This section introduces a performance model for parallel RMI on the simplest network configuration, local network. A threshold on the size of the computation is derived, which is used to rule out small `Job` objects when partitioning the application. The large `Job` objects above the threshold still need to be managed by the dynamic load balancing because the network traffic on the Internet is extremely dynamic.

The cost of a remote execution of a `Job` object consists of two parts: the round trip time of the method invocation and the computation time of the method on the remote host. The round trip time is dependent on the total data size of all the serializable objects in the `Job` object. In most computational applications, the computation time of the `Job` object mainly depends on the number of float-point operations and memory accesses in the method. It is also dependent on the execution rate of the JVM of the remote host.

It is difficult to predict the round trip time and the computation time in general. To build a model for these costs we have to make a lot of assumptions.

Firstly, we assume that the data size of the arguments objects is negligible compared with that of the mobile object itself. In computational applications, most data in mobile objects are floating-point data structures such as `double` arrays. The movement of these data between the hosts increases the round trip time of the RMI. The total number of floating-point data elements in the mobile object is denoted N .

3.1 Computation Time

Predicting the computation time of a method accurately is difficult. In general, the computation time can be divided into the times for floating-point operations (FLOP), memory accesses and the overhead of control structures such as loops. The total number of FLOPs depends on the algorithm of the method and is usually a function of size of the data array. The number of memory accesses is related to the number of operations and is heavily dependent on compiler optimizations used. Other architectural factors such as cache size can affect the memory access time significantly.

In this paper, we concentrate on those methods whose computation time can be predicted by a simple model based on the following assumptions:

- the number of floating-point operations is linear with the size of data arrays,

- the average number of memory accesses per floating-point operation is constant, and
- the overhead of control structures is small enough to be ignored.

Since the number of memory accesses per operation is constant, we can factor the memory access time into the time of floating operation. Assuming that each data element gives rise to K floating-point operations and the time of an floating-point operation including the time of memory accesses required is r_f , the computation time of a method is simply $K * N * r_f$, where N is the size of the data array.

FLOP	time	ratio
*	0.22 μ s	1.0
/	0.34 μ s	1.55
+	0.21 μ s	0.96
-	0.22 μ s	1.0
<code>sin(..)</code>	1.05 μ s	4.77
<code>sqrt(..)</code>	1.25 μ s	5.68
<code>log(..)</code>	1.47 μ s	6.68
<code>exp(..)</code>	19.42 μ s	88.27

Table 1. FLOP PERFORMANCE OF A JVM

Different floating-point operations including mathematical functions have different completion times. We can measure the ratio of floating-point operations against a canonical operation, say multiplication, and use the total number of canonical operations to quantify the computation time of a method. Table 1 shows the byte-code execution times of different types of float-point operations of Java Virtual Machine (JDK-1.1.4) on a DEC Alpha 255/233 machine. Each operation takes two memory accesses in the Java code. The results show that division in this JVM is 1.55 times as much as multiplication. A function `sin` can be counted as 4.77 multiplications and so on. Therefore, in our simple model, $K * N * r_f$, K is the number of canonical floating-point operations per data element and r_f the execution time of the canonical floating-point operation. r_f is also called *floating-point operation rate*.

3.2 Round Trip Time

The round trip time (RTT) of a remote method invocation is the overhead of the remote execution of the `Job` object. As described in Section 2, if the synchronous invocation is used, the trips in the both directions involve serialization and transmission of `Job` objects. If the mobile object contains large data arrays, the round trip time should increase proportionally.

We measured the round trip time of parallel RMI on a 100 MBits/s Ethernet by varying the array size of the mobile object from 0 to 6000. There are no floating-point operations in the method and, therefore, the computation time on the remote host is zero. The measurement is done on the local host which issues parallel RMI to $P = 1$, $P = 4$, or $P = 8$ identical remote hosts on the Ethernet. Both the local host and remote hosts are DEC Alpha 200/166 machines running JDK-1.1.4. A parallel RMI is implemented by concurrent threads on the local host. Barrier synchronization is used to time the completion time of a parallel RMI (see Section 4). Table 2 lists the average RTT of at least 10 parallel RMIs in all cases. The plots of these

array size	$P = 1$	$P = 4$	$P = 8$
0	88.8 ms	145.1 ms	258.3 ms
1000	95.5 ms	199.4 ms	344.0 ms
2000	108.9 ms	203.9 ms	429.6 ms
3000	125.7 ms	242.9 ms	532.9 ms
4000	146.2 ms	271.2 ms	610.7 ms
5000	156.1 ms	302.7 ms	683.0 ms
6000	175.7 ms	341.3 ms	761.1 ms

Table 2. RTT OF PARALLEL RMI

figures are shown in Figure 1. The measurement clearly shows that in every case

- the RTT is linear with respect to the size of the data array of the mobile object
- there is a constant overhead caused by the software layers from TCP/IP up to RMI. (88.8 ms in $P = 1$ case, 145.1 ms in $P = 4$ case, etc.)

Note that the RTT of 4 remote hosts is much less than 4 times of the RTT of a single remote host. This is because the interactions between the software layers on the remote hosts occur in parallel. The remote hosts mainly compete for the bandwidth of the local network for data transfer at the lowest data link layer.

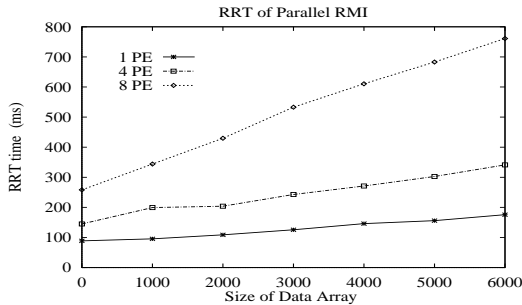


Figure 1. RTT OF PARALLEL RMI

We model the RTT of a single synchronous RMI as:

$$T_{rrt} = 2 * (t_{const} + N * r_t)$$

where

- t_{const} represents the constant overhead of a RMI,
- N is the number of `double` data elements contained in the mobile object, and
- r_t denotes the *data transmission rate* over the net.

Recall that the synchronous `push(Job job, ...)` returns a `Job` object (see Section 2). There are data transmissions in the both directions and this is the reason for the coefficient 2 in the formula above. Let t_{const}^P and r_t^P be the t_{const} and r_t , respectively, for the case of P remote hosts. We use a simple model¹ to obtain the values of t_{const}^P and r_t^P shown below:

$$t_{const}^P = \frac{RTT_P(0)}{2 * P}$$

$$r_t^P = \frac{RTT_P(6000) - RTT_P(0)}{2 * 6000 * P}$$

where $RTT_P(N)$ denotes the RTT with array size N in the case of P remote hosts. The results calculated from Table 2 are listed in Table 3.

	$P = 1$	$P = 4$	$P = 8$
t_{const}	44.4 ms	18.14 ms	16.14 ms
r_t	7.24 μ s	4.09 μ s	5.24 μ s

Table 3. t_{const} AND r_t OF A SINGLE RMI

The local host should know the number of the remote hosts registered for the meta-computing. While this number, P , may change at the run-time, the local host can always measure the RTT on the fly and calculate t_{const}^P and r_t^P , accordingly.

3.3 Speedup of Parallel RMI

Let $(t_{const}^P + N * r_t^P)$ be denoted as:

$$t_d = t_{const}^P + N * r_t^P \quad (1)$$

Then the total completion time of a single RMI is

$$T = 2 * t_d + t_c$$

where t_c is the computation time on the remote host.

If P `Job` objects are executed at the local host, the sequential completion time is $P * t_c$ assuming the local JVM has the same FLOP rate as the remote JVMs.

Now we model the completion time of the parallel RMI on P remote hosts. First of all, the minimum completion time is $2 * P * t_d$, because, with $t_c = 0$, the

¹A better model is to use the least square method for approximation, but our simple model seems sufficient for our purposes.

RTT of the parallel RMI is $2 * P * t_d$. Secondly, the computation time t_c can be overlapped with each other as well as with t_d , because the computations are carried out in the remote hosts independently. Figure 2 illustrates a simplistic model of parallel RMI ($P = 4$) where each individual RMI is modeled by a t_d followed by a t_c and another t_d . All the RMIs are assumed to start at the same time. In Figure 2, t_d and t_c are represented by thick lines and rectangles, respectively. Thick lines cannot be overlapped with each other because they compete for the same network. The dashed lines represent the times caused by delays.

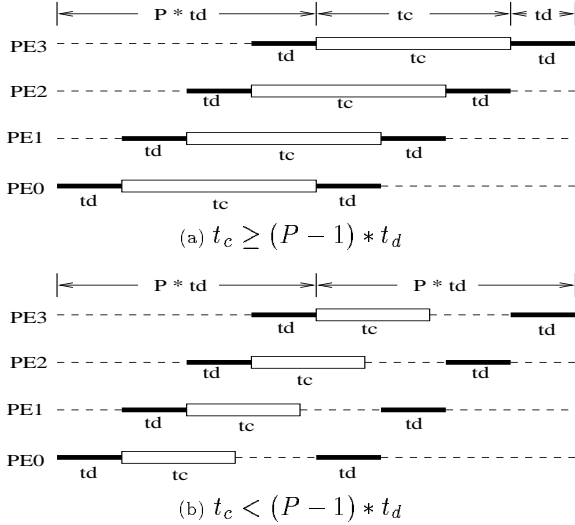


Figure 2. PERFORMANCE MODEL OF PARALLEL RMI

If $t_c \geq (P - 1) * t_d$ as the case shown in Figure 2(a), the t_c of the last remote host can be overlapped with all the t_d following the corresponding t_c of the other $(P - 1)$ remote hosts. Therefore, the parallel completion time is $(P * t_d + t_c + t_d)$. When $t_c = (P - 1) * t_d$, the parallel completion time $(P * t_d + t_c + t_d)$ is reduced to the minimum completion time $2 * P * t_d$. If $t_c < (P - 1) * t_d$, the parallel completion time remains to be $2 * P * t_d$ as shown in Figure 2(b). That leads to the model for the speedup of parallel RMI, denoted S_p , as follows:

$$S_p = \begin{cases} \frac{P * t_c}{(P + 1) * t_d + t_c} & \text{if } t_c \geq (P - 1) * t_d \\ \frac{t_c}{2 * t_d} & \text{otherwise} \end{cases} \quad (2)$$

When $t_c = (P - 1) * t_d$, $S_p = (P - 1) / 2$, a little less than the half of the number of remote hosts used. With t_c increases, the speedup increases, approaching P . If $t_c < (P - 1) * t_d$, the speedup is less than $S_p = (P - 1) / 2$.

3.4 Half-Point Threshold

Let us call the situation of $t_c = (P - 1) * t_d$ the *half-point*. To have a speedup larger than $(P - 1) / 2$ of the

half-point, the computation time t_c on the remote host should be at least $(P - 1) * t_d$.

According to our model, the computation time is

$$t_c = K * N * r_f \quad (3)$$

Recall that t_d is also dependent on N (see (1)). Given N and r_f , K should be sufficiently large in order to gain the half-point speedup. Putting it all together, we can derive that K must be at least

$$K_0 = (P - 1) * \left(\frac{t_{const}^P}{N * r_f} + \frac{r_t^P}{r_f} \right) \quad (4)$$

in order to gain a speedup higher than the half-point.

K_0 is called the *half-point threshold*. It can be used to rule out many mobile objects unsuitable for meta-computing. For example, if we try to partition and decompose the red-black parallel Successive Over-Relaxation (SOR) algorithm [12] to mobile objects for the meta-computing, the number of floating-point operations per data element, K , is going to be very low. Since the parallel blocks need to exchange their data near the boundaries after each iteration of the iterative loop, the method for the mobile object for each block is restricted to the computation of one iteration. If the 4-point stencil is used for the SOR, there are only 3 additions, 1 division and 5 memory accesses for each data element. This is equivalent roughly to 4.43 multiplications according to Table 1. Obviously, such mobile objects will never be able to reach the half-point speedup in any circumstances. We should rule out the SOR problem for meta-computing.

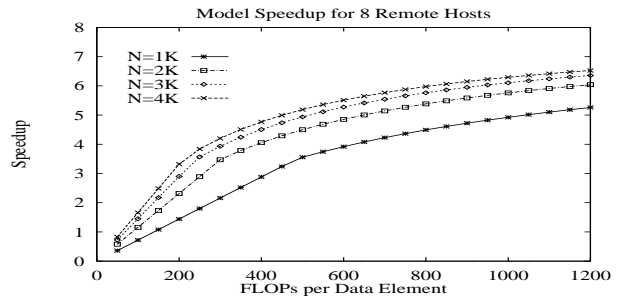


Figure 3. MODEL SPEEDUP FOR 8 REMOTE HOSTS

Let us use the figures from Table 3 to model the speedup of parallel RMI when $P = 8$. The measured FLOP rate, r_f , of these identical remote hosts (DEC Alpha 200/166 running JDK-1.1.4) is $0.308 \mu s$ per element. The speedup calculated by the model in (2) for different K when the data sizes are 1000, 2000, 3000 and 4000 are plotted in Figure 3. According to (4), the half-point speedup is reached at $K_0 = 485$, $K_0 = 302$,

$K_0 = 242$ and $K_0 = 210$ for these data sizes, respectively. These thresholds are summarized in Table 4.

array size	Threshold
1000	485
2000	302
3000	242
4000	210

Table 4. THRESHOLD FOR HALF-POINT SPEEDUP

4 Experiments

To evaluate the performance model described above, we measured the actual completion time of parallel RMI with 8 remote hosts. Both the local host and remote hosts are DEC Alpha 200/166 machines running JDK-1.1.4 on a 100 Mbits/s local Ethernet. The mobile object wrapped up in the `Job` objects contains a one-dimensional `double` array of size N . The method simply performs K multiplications on each data element. The code of this mobile object is as follows:

```
public class OneD implements Serializable {
    private double data[];
    private int size, flop;

    OneD(int sz, int fl) {
        size = sz; flop = fl;
        data = new double[size];
        for (int i = 0; i < size; i++)
            data[i] = 100.0 * i;
    }

    public void work() {
        for (int i = 0; i < size; i++)
            for (int j = 0; j < flop; j++)
                data[i] *= 0.99999;
    }
    ...
}
```

For the parallel synchronous RMI, P threads are created at the local host. Each thread makes repeated synchronous RMI calls to the remote host. The main part of the `run()` method of these threads is as follows:

```
for (int i = 0; i < batch; i++) {
    try {
        rtn = sl.push(job, ...);
    } catch (RemoteException e) { }
    try {
        barrier.meet();
    } catch (InterruptedException e) { }
}
```

Here, `sl` is the remote object that implements the remote method `push(Job, ...)` shown in Section 2.

`job` and `rtn` are the original and returned `Job` objects, respectively. `barrier` is an object for barrier synchronization[13]. Its `meet()` method won't return until all the threads have called this method. `batch` is the number of repeated tests. Each test is a parallel RMI to P remote hosts. The `for` loop is timed and the total execution time divided by the value of `batch` gives the average completion time of the parallel RMI calls. The value of `batch` in our experiments is 14.

We measured the parallel completion times for the different combinations of the array size, N , and the number of multiplications per data element, K .

The sequential completion time is calculated by $P * N * K * r_f$ as if the P mobile objects were all computed by a single host. The r_f for the multiplication on the machines we used is $0.308 \mu s$ per element.

The actual speedup of is obtained by dividing the calculated sequential completion time by the measured parallel completion time. The results are plotted in Figure 5.

In order to compare the actual speedup with the model speedup, we re-plotted them for each different array size as shown in Figure 4.

The experiments show that our performance model matches well the actual speedup when K is less than 500. As K gets larger, the actual speedup becomes less predictable and tends to be below the predicted speedup. The actual speedup when $N = 1000$ matches well the model for all K between 50 and 1200. This suggests that our model is precise for the small tasks with the total number of multiplications less than around 1,500,000. We believe that this is because longer the computation task, the more the actual computation time tends to fluctuate. Our model assumes that all computation times for the same job on the remote hosts are the same. In the experiments, the parallel completion time is actually the maximum time if the completion times of individual RMIs are different.

In any case, our model predicts the half-point speedup and, hence, the half-point threshold, reasonably well.

5 Conclusion and Future Work

We have presented a meta-computing program model based on the concept of mobile objects and its implementation in Java. A simple performance model of parallel RMI for mobile objects was presented. From this performance model, the job size threshold for the speedup close to the half of the number of the remote hosts can be calculated. This threshold can be used to rule out many applications for meta-computing if the job size of their mobile objects is below the threshold.

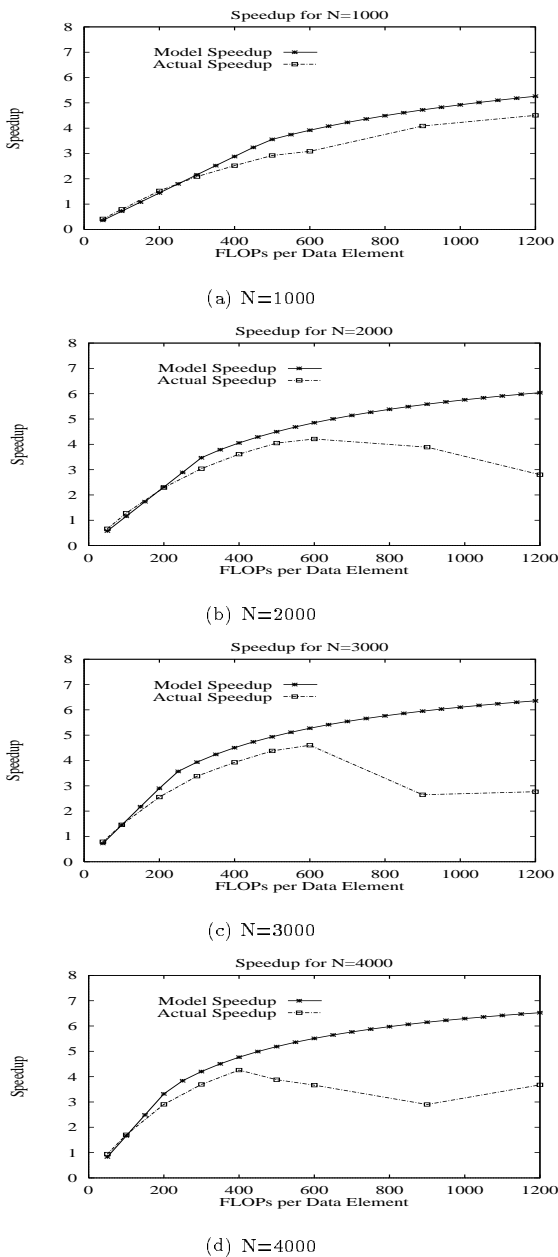


Figure 4. COMPARISONS OF SPEEDUPS

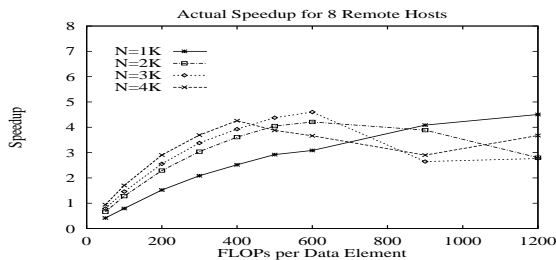


Figure 5. ACTUAL SPEEDUP FOR 8 REMOTE HOSTS

The experimental results of the performance model are also presented. The results show that our performance model predict the actual performance reasonably well around the threshold.

As we pointed in the paper, the actual round trip time and computation time of a RMI on the Internet is very difficult to predict. We believe that for the balanced load distribution, dynamic load balancing of mobile objects should be used.

We also believe that the regular measurement of the round trip time and the FLOP rate of the remote hosts is necessary to guide the adaptive dynamic scheduling and load balancing at the run-time. We plan to work on this issue in the future.

References

- [1] Tommy Thorn. Programming languages for mobile code. *ACM Computer Surveys*, 29(3):213–239, September 1997.
- [2] James Gosling, Bill Joy, and Guy Steele. The java language specification. <http://java.sun.com/docs/books/jls/html/index.html>, 1996.
- [3] A. Alexandrov, M. Ibel, K.E. Schauser, and C. Scheiman. SuperWeb: Research issues in java-based global computing. *Concurrency: Practice and Experience*, June 1997.
- [4] David Abramson, Ian Foster, Jon Giddy, Andrew Lewis, et al. The Nimrod computational workbench: A case study in desktop metacomputing. In *Proceedings of the 20th Australasian Computer Science Conference*, pages 17–26, February 1997.
- [5] J.-P. Billon. Java security: Weakness and solutions. http://www.dyade.fr/actions/VIP/JS_pap2.html, 1997.
- [6] J.E. Baldeschiwiler, R.D. Blumofe, and E.A. Brewer. ATLAS: An infrastructure for global computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for WorldWide Applications*, 1996.
- [7] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th Conference on Parallel and Distributed Computing Systems*, 1996.
- [8] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards worl-wide supercomputing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for WorldWide Applications*, 1996.
- [9] N. Camiel, S. London, N. Nisan, and O. Regev. The POP-CORN project: Distributed computation over the internet in java. In *Proceedings of the 6th International World Wide Web Conference*, April 1997.
- [10] Bernd O. Christiansen, Peter Cappello, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser, and Daniel Wu. Javelin: Internet-based parallel computing using java. In *Proceedings of 1997 ACM Workshop on Java for Science and Engineering Computation*, June 1997.
- [11] Sun Microsystems Inc. Java home page. <http://java.sun.com>, 1996.
- [12] Geoffrey C. Fox, Mark A. Johnson, et al. *Solving Problems on Concurrent Processors*, volume 1, General Techniques and Regular Problems. Prentice-Hall, 1988.
- [13] Paolo A.G. Sivilotti and K. Mani Chandy. Reliable synchronization primitives for java threads. Technical report, Department of Computer Science, California Institute of Technology, 1996.