FI MU

Faculty of Informatics
Masaryk University

High Performance Computing in JAVA – Fact or Fiction?

by

Václav Dvořák
Lubomír Markovič

FI MU Report Series **FIMU-RS-2000-11**

 December 2000

High Performance Computing in JAVA – Fact or Fiction ?

Václav Dvořák
Brno University of Technology
Faculty of Electrical Engineering
and Computer Science
Brno, Czech Republic

Lubomír Markovič[1]
Masaryk University
Faculty of Informatics
Brno, Czech Republic

The intent of this article is to give an overview of possibilities for building distributed/parallel applications in the JAVA language and to judge the suitability of some of them for high performance computing. In the first part different approaches and libraries are presented and in the second part of this article, RMI, HORB and CORBA libraries are compared to direct communication over sockets. Gauss–Jordan algorithm for computation a system of linear equations was used to test these libraries.

Overview of basic approaches

Different approaches for building parallel/distributed applications in JAVA are described in next sections.

Threaded Approach

Java has the build-in support for threads. There are usual constructs for manipulating threads (fork, join, suspend, ...) too. The threaded approach is used mainly to improve the latency of programs. Dividing the high

[1]Supported by Czech Grant Agency, contract no. GAČR 201/98/0532.

performance-computing problem into several threads has the sense only for multi-processor machines, because on single-processor machine the threads must share the one processor with some additional overhead for thread's alternation.

Networks of Workstations Approach

There is a natural attempt to have the development of distributed applications as easy as possible on one side and use the power of all computers in clusters (or some net) on the other side. There exist libraries that try to join these two attempts. Shortly speaking, if you have a thread-based application and transform it in some ,,easy" way this systems (NoW) are able to run some threads on different computers in predefined parallel virtual machine. Typically this system and its libraries must simulate the shared memory over the computers in virtual machine and there must be also some new language keywords for specifying the remote-able threads, so some special preprocessor must be included in these systems. We can name two libraries: JavaParty and JavaNOW. JavaParty is publicly available on the internet [6], JavaNOW can be obtained by contacting the author [7].

Socket Based Communication Approach

There is a support for a basic communication over sockets in the standard Java's package. The communication may be realized through TCP (Transmission Control Protocol) or UDP (User Datagram Protocol). TCP offers reliable environment, where we are sure that data will be send correctly, but its overhead is bigger. UDP doesn't guarantee the correct transport of data, but its overhead is minimal. In our example (described below) we have used the UDP, but there is a strong suggestion of a well-balanced system, where each server is listening when someone is sending some data, otherwise data will be lost.

MPI Approach

MPI (Message Passing Interface) is a standard of MPIF (Message Passing Interface Forum) for communication between applications based on message passing approach. MPI standardizes the interfaces of libraries for message passing. There are implementations of this standard in C, C++ and there

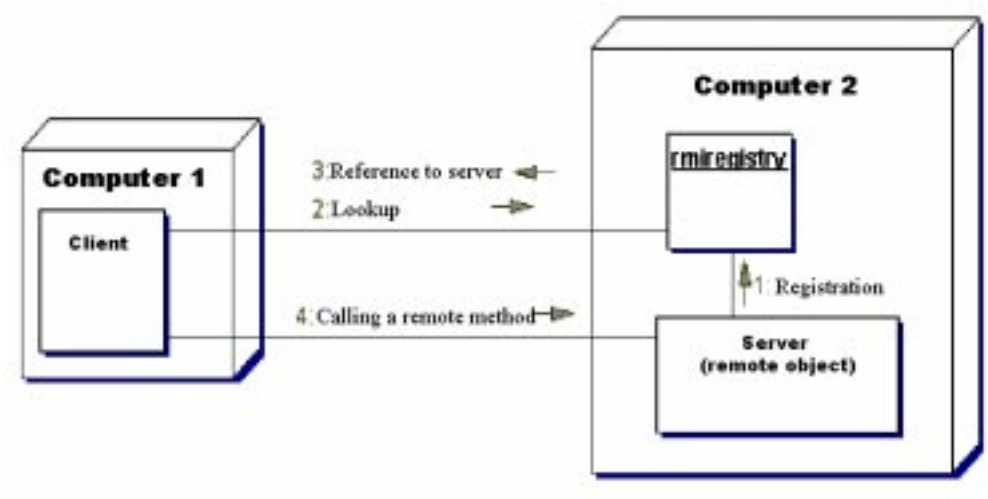


Figure 1: Basic RMI application structure

is also a natural attempt to translate it to the JAVA as well. There are three main approaches how to do it. The first way tries to implement this standard in the pure JAVA, but the libraries' performance is very small. The second way tries to wrap existing C or C++ libraries by JAVA code using NMI (Native Method Invocation) but these JAVA libraries are portable as well as the underlying native library. The last approach extends the JAVA Virtual Machine Specification so special JAVA compilers and interpreters are needed. The performance is, thanks to special JAVA instructions (and new language keywords) better but the portability is disputatious.

RMI Approach

RMI (Remote Method Invocation) is the JAVA standard for executing methods of remote objects. Sometime it's called as a JAVA version of RPC (Remote Procedure Call). RMI enables any application (client) to call the methods of a remote server, this server is in fact a remote object and the client can work with it as it's a local object. It's achieved by proxy objects that hide the needed communication to the client. The client can gain information needed to contact the server by asking the special name-service (`rmiregistry`) program that must run on every computer where some RMI server runs. The typical application's structure is shown in the picture 1. Details of RMI standard are behind the scope of this article and can be acquired in the sources listened in the references (especially [5]).

HORB Approach

HORB is the JAVA library that provides the same things as RMI does and something more. This library is developed and maintained by Dr. Hirano Satoshi, Electrotechnical Laboratory and NJK Corporation. The role of `rmiregistry` plays the `horb` (HORB daemon) here. In this approach we can call methods of a remote object that is not just running. The HORB daemon can create a new instance of this remote object (this approach is called "generation model" in opposite to "connection model" in RMI).[2] HORB provides the connection model as well. Shortly speaking we can say that implementation of distributed applications in HORB is more straightforward than in RMI. More details can be found for example in [4].

CORBA Approach

CORBA (Common Object Request Broker Architecture) is the standard of OMG (Object Management Group) for heterogeneous distributed applications based on object-oriented paradigm. It's very complex standard and a detailed description is beyond the scope of this article (specifications can be found for example in [8]). For our purposes it's important that the architecture of the program is very similar to RMI or HORB version. The only significant difference is that there is no `rmiregistry` or `horb demon` on each computer where some server resists but there is only one `nameserver` that provides for all servers nearly the same things as the previous ones. The typical application structure is shown in the picture 2.

EJB Approach

We must mention another approach yet to have a complete picture. Sun's Enterprise Java Beans (EJB) component framework provides services for transactions, security, persistence, ... The core of this approach is based on creating objects that are incorporated into EJB environment. This environment provides standardized interfaces to many services so the developer needn't to self-develop many often-used important services but can just use them. This approach provides robust environment for distributing computing maximally simplifying creating new enterprise objects (remote servers).

[2] This feature is also included in RMI since 1.2 version

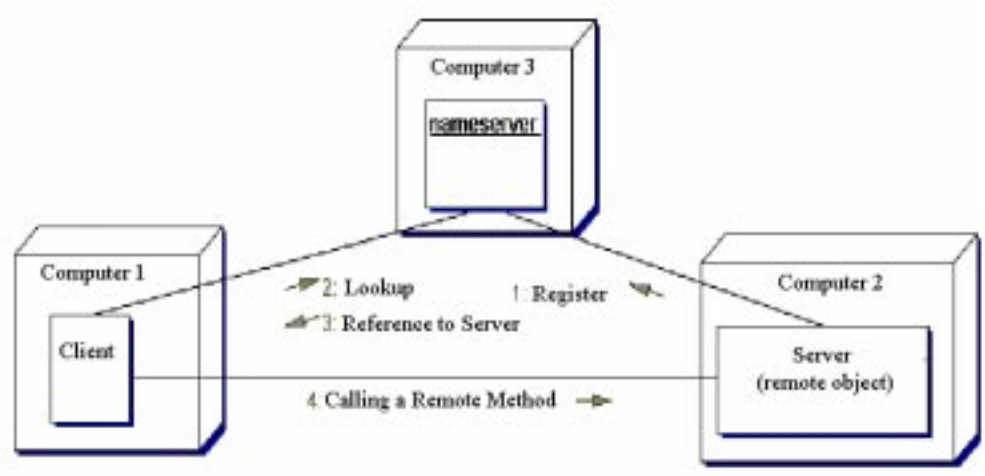


Figure 2: Basic CORBA application structure

Measurement of Performance

The main purpose of this article is to judge the suitability of RMI, HORB and CORBA approaches for high performance computing, because they are the most commonly used for distributed applications and provides good programming environment. A demonstrating application was created for this approaches and their efficiency was compared to each other and mainly to the application based only upon UDP sockets. The motivation is to show whether these libraries can substitute, sometimes hardly to program, direct communication over the sockets. The programmer needn't to deal with direct-net communication using these libraries and can concentrate themselves to a problem he solves.

Computation Environment Used

All computers used in this experiment had the following configuration:

Hardware: 10 Sun Ultra 5, UltraSparc IIi 270 MHz, 64 MB RAM

Software: Solaris 2.6

Router: Extreme Networks Summit 48/L3, 2x1000BASE-SX, 48x10/100 BASE-T

JAVA: version 1.2.2

HORB: version 2.0

ORBacus: version 3.3

Algorithm for Solving a System of Linear Equations

An application used to measure the performance was the implementation of algorithm for solving a system of linear equations. We describe the parallel version of this algorithm at first.

Let's have a squared matrix $A(n \times n)$ with elements a_{ij} and a vector $v(n)$ of right sides. Our goal is to transform the matrix to have nonzero values only on the diagonal. We can use the Gauss-Jordan algorithm that is implemented by the following code:

for i:=1 to n do
    for j:=1 to n, $(j \neq i)$ do
        v_j:=v_j-$v_i a_{ji}/a_{ii}$
        for k:=1 to n do
            a_{jk}:=a_{jk}-$a_{ik}a_{ji}/a_{ii}$

This algorithm can be parallelized by dividing the matrix (and vector as well) into several (equally big) pieces. Each of these pieces will be maintained and computed by different server (thread, process). The only thing this servers need to inform each other is the pivot row. So always the only one server is the owner of the pivot row at every moment during this program's run and this server (we will call it "owner") must send the pivot row to the other ones (figure 3). It's also useful to realize that it's not needed to send the whole row all the time, because some number of first elements in the pivot row will have to have the value of zero. It enables us to optimize the communication.

Applications' Structure

In RMI, HORB and CORBA version there are separate servers for each part of the matrix. Each server supports the following interface:

```
module Equations {
```

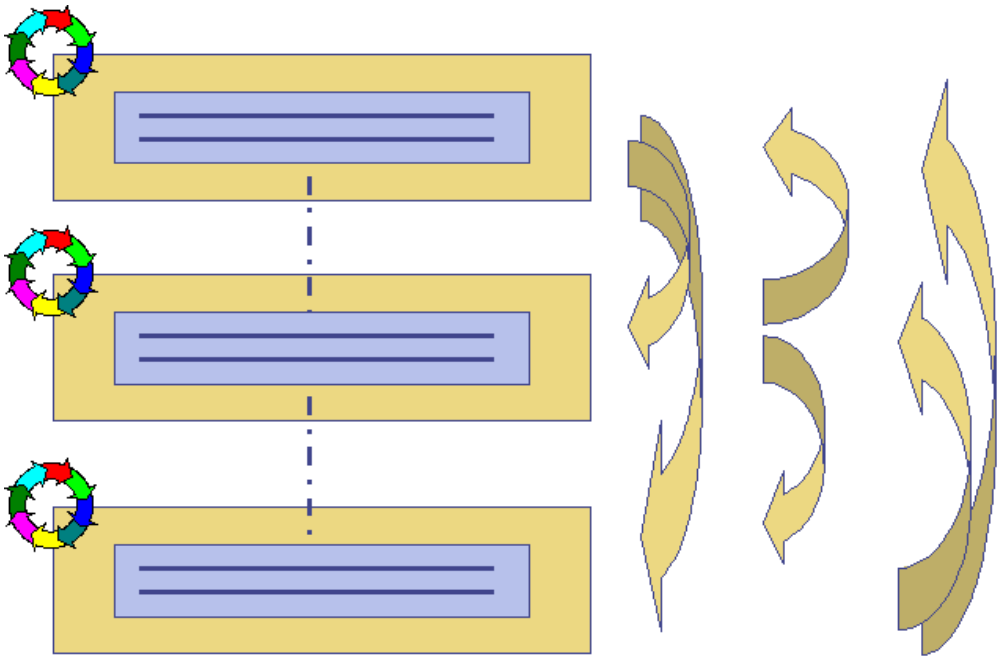


Figure 3: Parallelization of the algorithm

```
  typedef sequence<string> stringSeq;
  typedef sequence<float> floatSeq;

 interface EqSolver {
  void setParams(in long Id, in stringSeq Addresses,
                 in long MatrixDim);
  void start();
  void putPivot(in long PivotId, in floatSeq Row);
 };

};
```

`setParams` method serves to inform the server about the dimension of the matrix being solved and about the locality of other cooperating servers by specifying their names in `Addresses` parameter). Parameter `Id` says what's ,,my" index. When this method is called, the server initializes the adequate part of matrix (by random values) and prepares communication channels to other servers (by contacting appropriate name-server to gain references to them).

`start` method starts the computation itself.

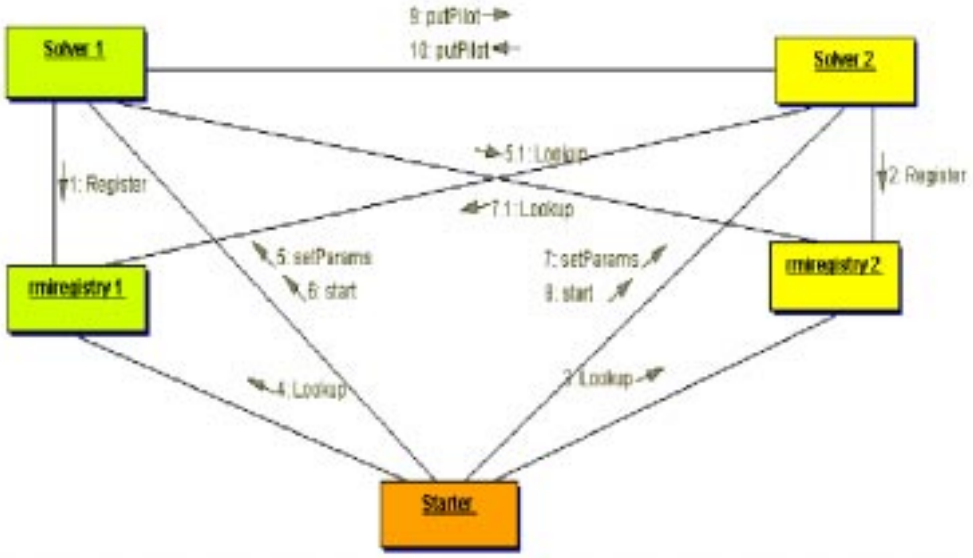


Figure 4: RMI application structure

When the owner (server that has the pivot row) wants to send the pivot to others, it uses the `putPivot` method to do it (`PivotId` says who is sending and `Row` is the pivot).

We need also another program (`Starter`) that is responsible for setting all servers by calling the `setParams` method and for starting the computation by calling the `start` method for all servers. The `Starter` is needed only for initial synchronization of servers (we must assure that all servers are registered in name-server when others are asking for them). The starter can be omitted if it's done in some other way (for example by some adequate lag). The whole application structures for RMI and CORBA are shown in figures 4 and 5. In socket version we have a separate server for each part of the matrix as well, but there is no explicit interface. Two threads run in each server, one for computation (as well as in previous ones) and one for listening the communication channel (it's created automatically for each communication in previous approaches). There must be the first synchronization communication step before the computation can begin to assure that every server is properly instantiated. The `Starter` plays this role. The UDP (User Datagram Protocol) was used. The basic structure is shown in figure 6).

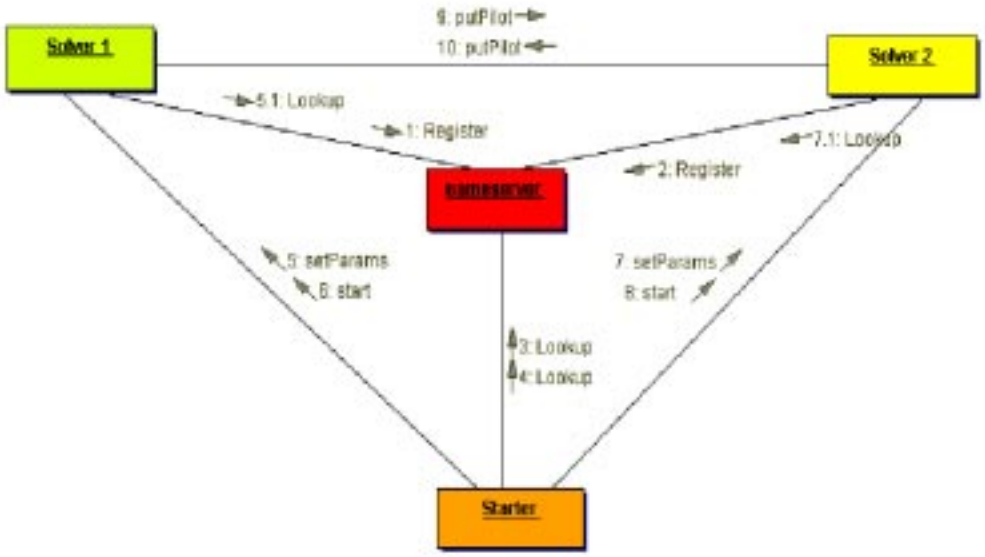


Figure 5: CORBA application structure

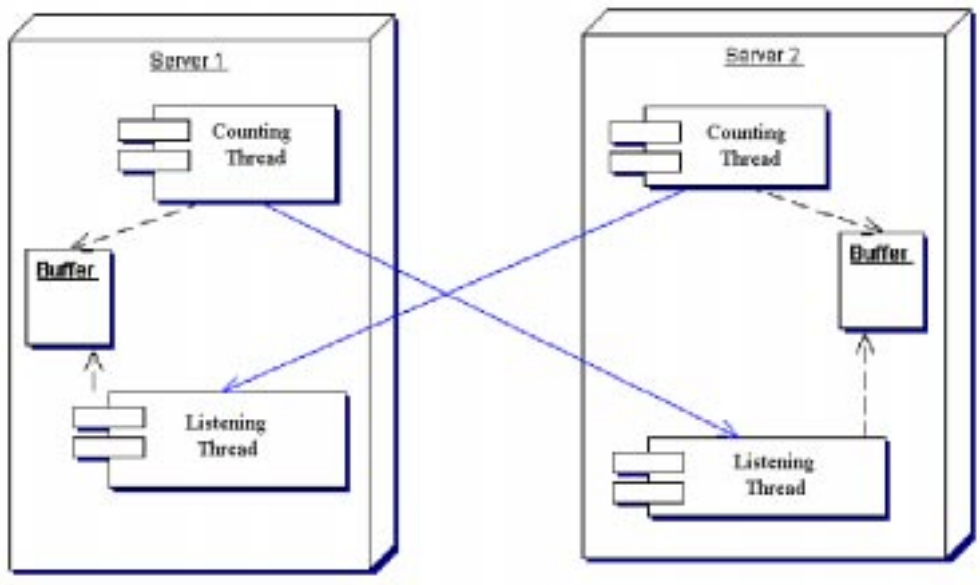


Figure 6: Socket application structure

Way of Measurement

All remote calls are synchronous (except the socket version), it means, the caller need to wait until the receiver receives the message (pivot here). All versions were implemented as non-blocking, it means, in all servers there is a buffer to store incoming messages, so the caller needn't to wait until the receiver is in the stage of computing when it needs the message (pivot row).

The measurement was done by calling `System.currentTimeMillis()` function of Java package. It returns the system time with the precision in milliseconds. This function was called at the beginning and the end of the computation (call of the `start` method), so the time of instantiating the matrix (or the appropriate piece of matrix) and communication channels was not included. Of course it's not very precise way to do some measurement, because it's hardly dependant on well-balanced system, where only a few other processes are running to influent the time minimally.

The measurement was done on matrixes $(400x400), (800x800), (1600x1600)$ that were computed by $1, 2, 4, 8$ servers.

Way of Computing The Performance

The speedup was computed by the next equation:

$$S_p = T_s/T_p$$

Where T_s is the time of one-process version (it's the sequential time) and T_p is the time of p-process version. The T_s should be the time of the best known sequential algorithm, but for our purposes it's not so important.

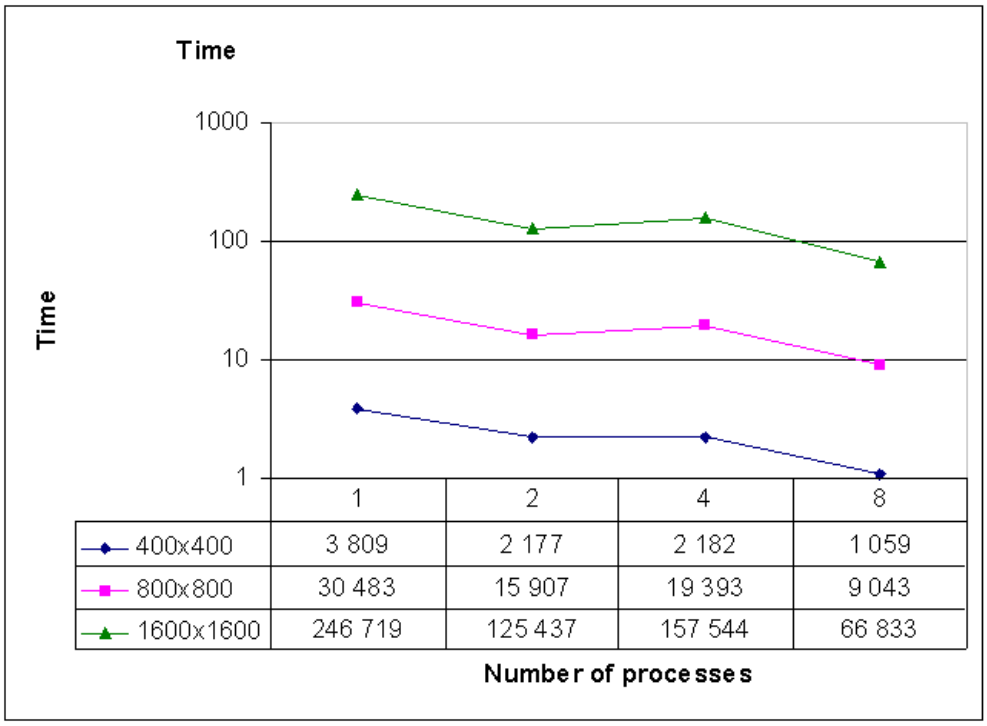
The efficiency was computed by this equation:

$$E = S_p/p$$

The efficiency shows us the average process(or) utilization.

Socket Version

The communication over the protocol UDP was used in this version. I repeat again that there is a strong suggestion of the same computation speed of clients in UDP version. Every non-owner client must listen when the owner is sending the data, otherwise the data will be lost and synchronization of



	1	2	4	8
400x400	3 809	2 177	2 182	1 059
800x800	30 483	15 907	19 393	9 043
1600x1600	246 719	125 437	157 544	66 833

Figure 7: Socket version – Time

clients will fail. The results are in the next graphs: figure 7, 8 and 9 (times are in miliseconds).

RMI Version

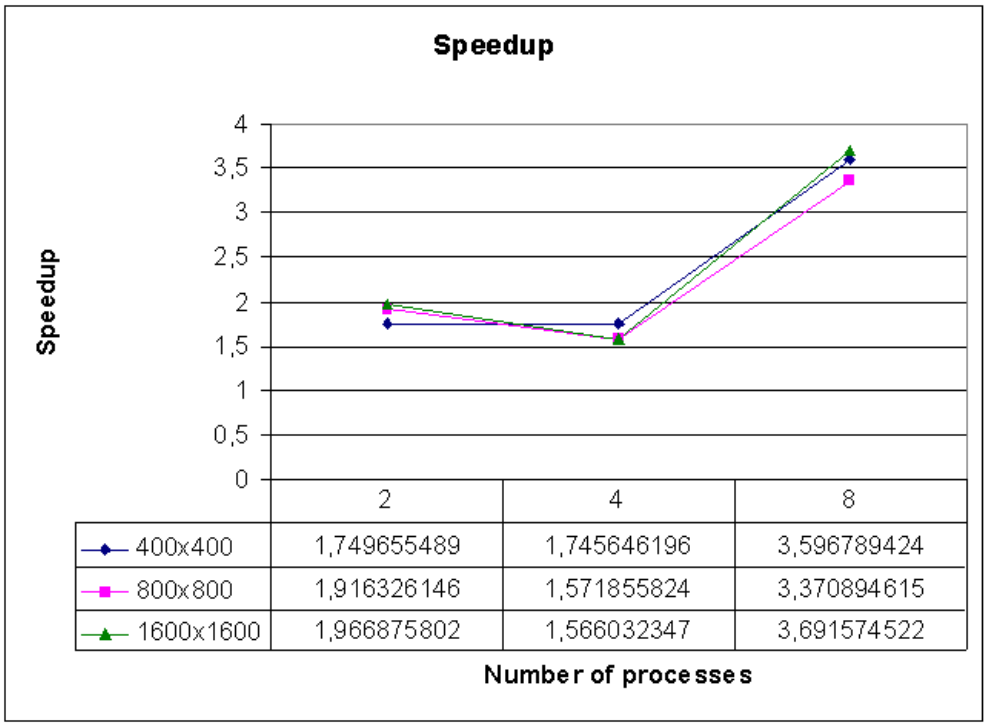
Results for this version are in the next graphs: figure 10, 11 and 12 .

HORB Version

Results for this version are in the next graphs: figure 13, 14 and 15 .

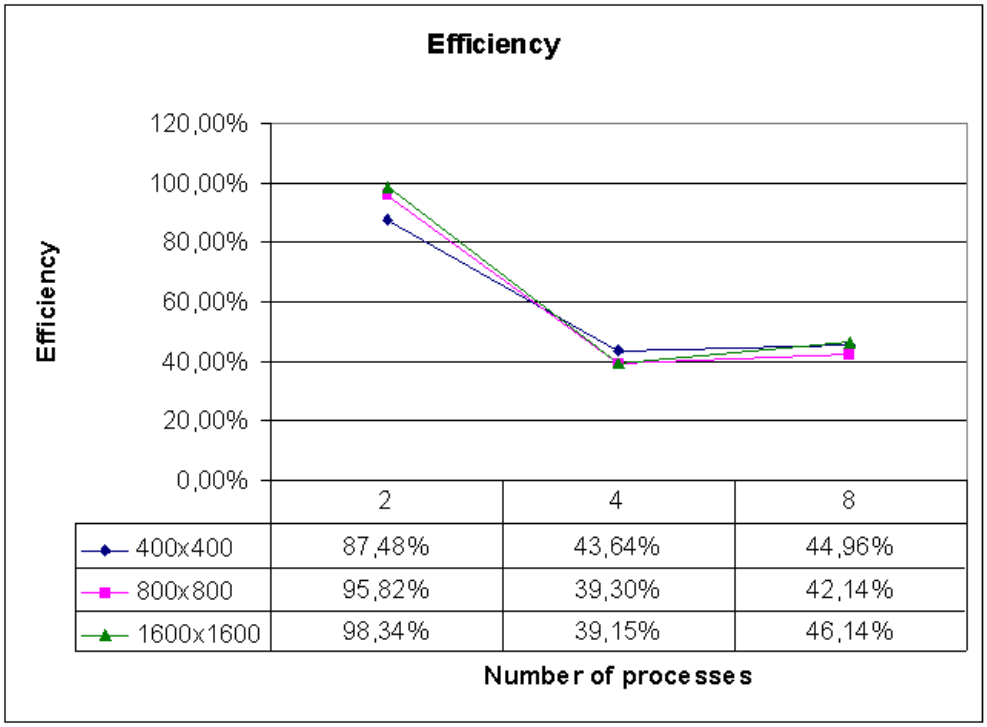
CORBA Version – ORBacus

The implementation of CORBA named ORBacus from O.O.C. Inc. was used. Results for this version are in the next graphs: figure 16, 17 and 18 .



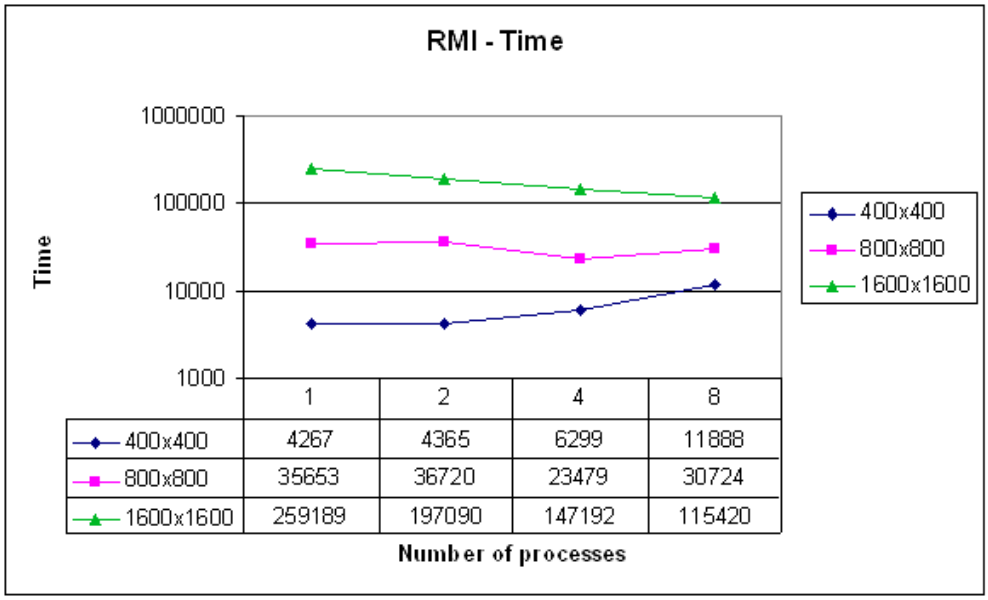
	2	4	8
400x400	1,749655489	1,745646196	3,596789424
800x800	1,916326146	1,571855824	3,370894615
1600x1600	1,966875802	1,566032347	3,691574522

Figure 8: Socket version – Speedup



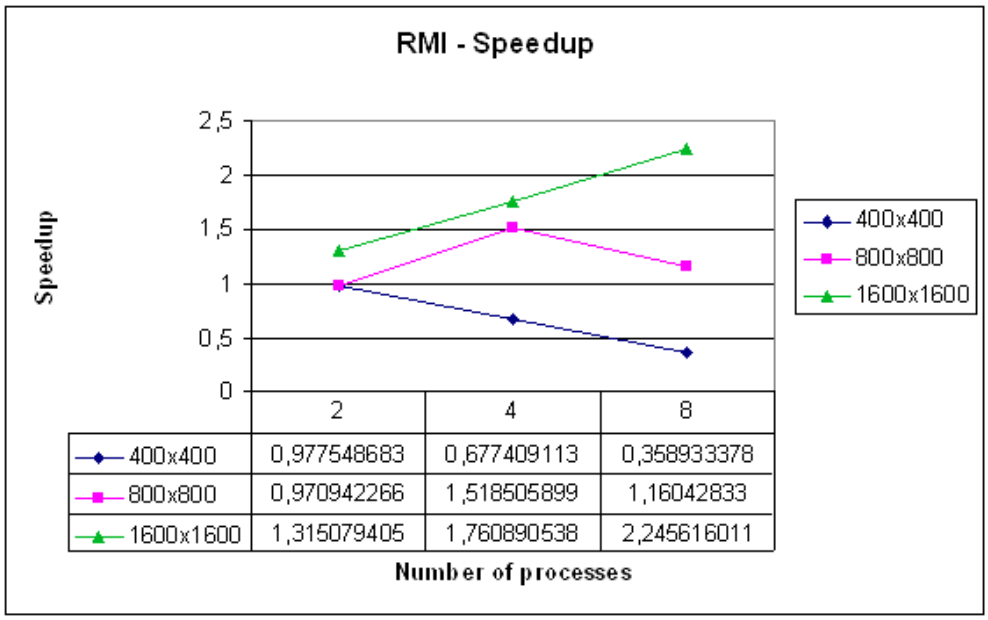
	2	4	8
400x400	87,48%	43,64%	44,96%
800x800	95,82%	39,30%	42,14%
1600x1600	98,34%	39,15%	46,14%

Figure 9: Socket version – Efficiency



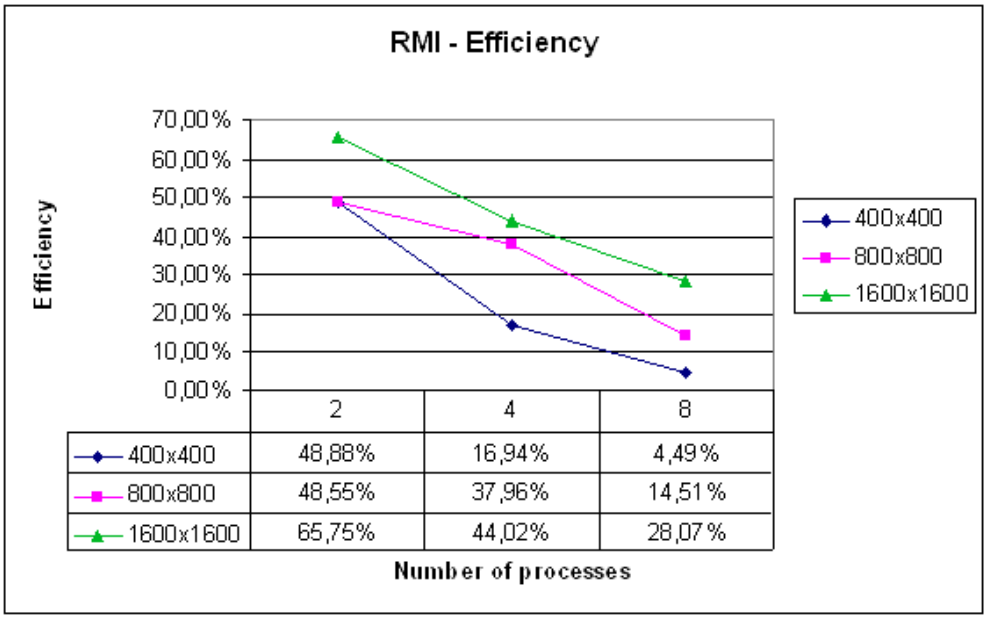
	1	2	4	8
400x400	4267	4365	6299	11888
800x800	35653	36720	23479	30724
1600x1600	259189	197090	147192	115420

Figure 10: RMI version – Time



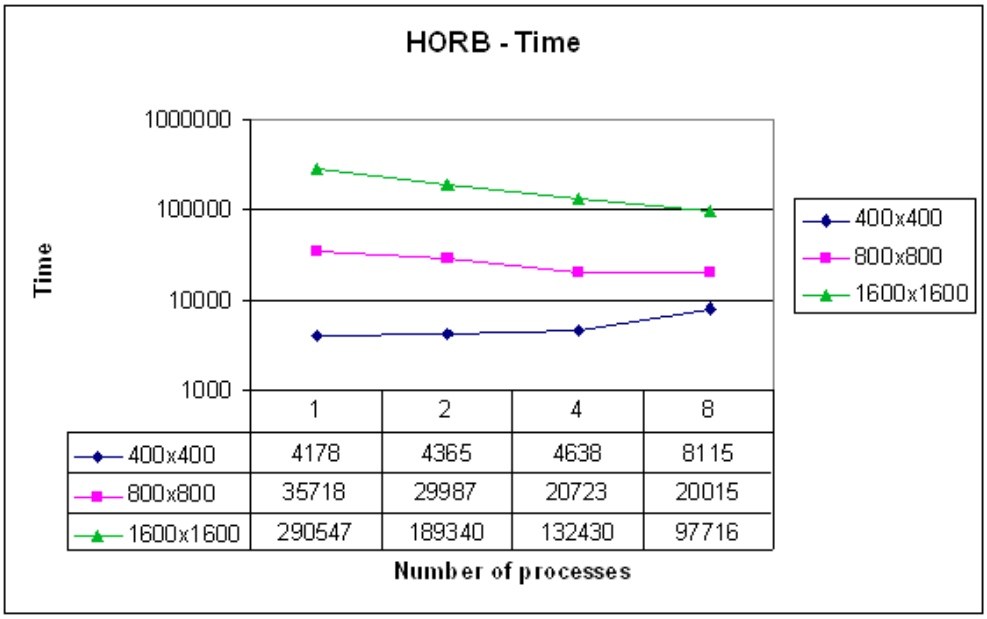
	2	4	8
400x400	0,977548683	0,677409113	0,358933378
800x800	0,970942266	1,518505899	1,16042833
1600x1600	1,315079405	1,760890538	2,245616011

Figure 11: RMI version – Speedup



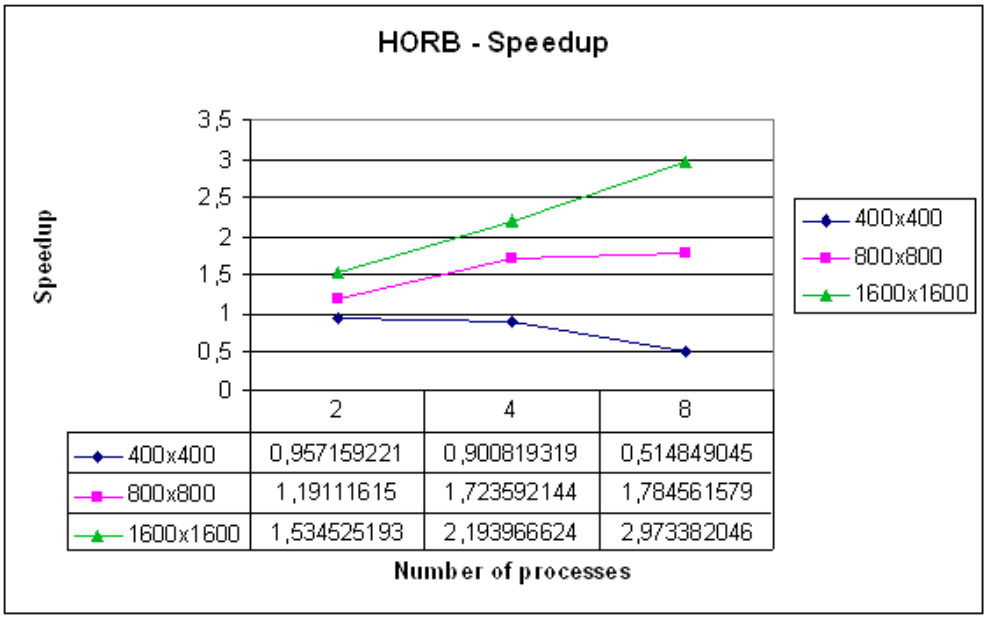
	2	4	8
400x400	48,88%	16,94%	4,49%
800x800	48,55%	37,96%	14,51%
1600x1600	65,75%	44,02%	28,07%

Figure 12: RMI version – Efficiency



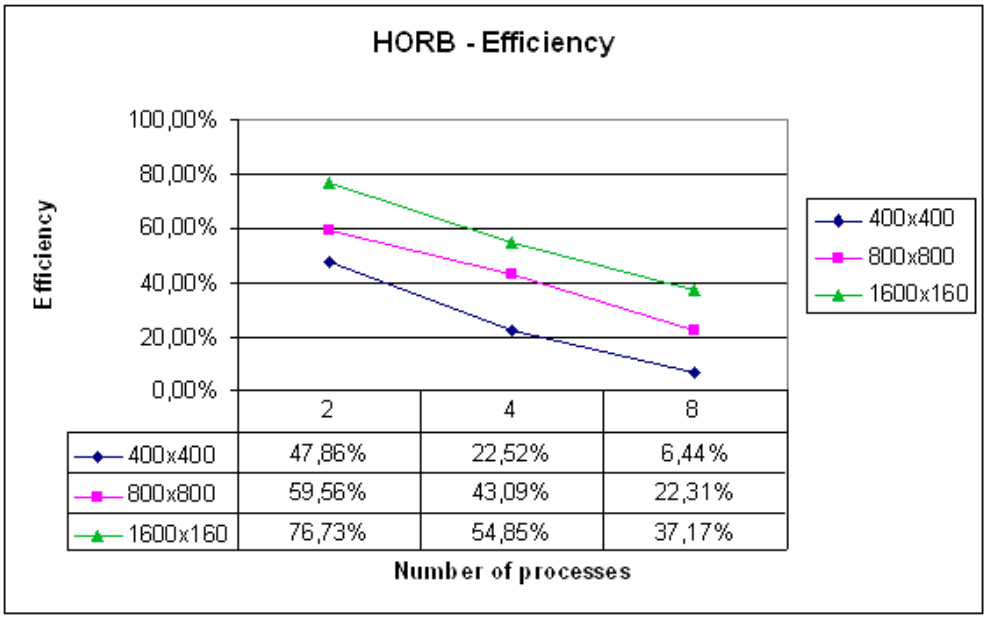
	1	2	4	8
400x400	4178	4365	4638	8115
800x800	35718	29987	20723	20015
1600x1600	290547	189340	132430	97716

Figure 13: HORB version – Time



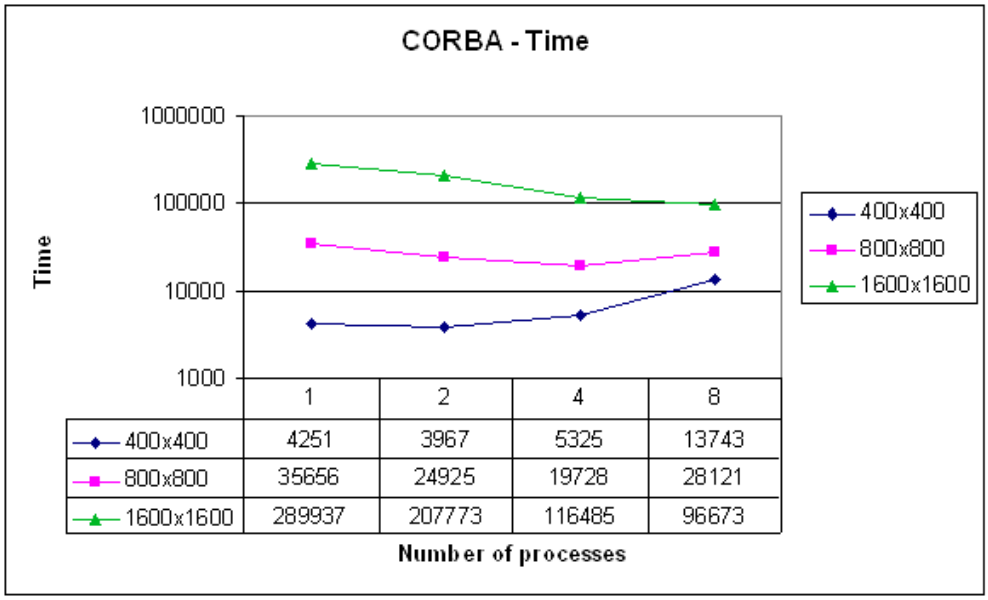
	2	4	8
400x400	0,957159221	0,900819319	0,514849045
800x800	1,19111615	1,723592144	1,784561579
1600x1600	1,534525193	2,193966624	2,973382046

Figure 14: HORB version – Speedup



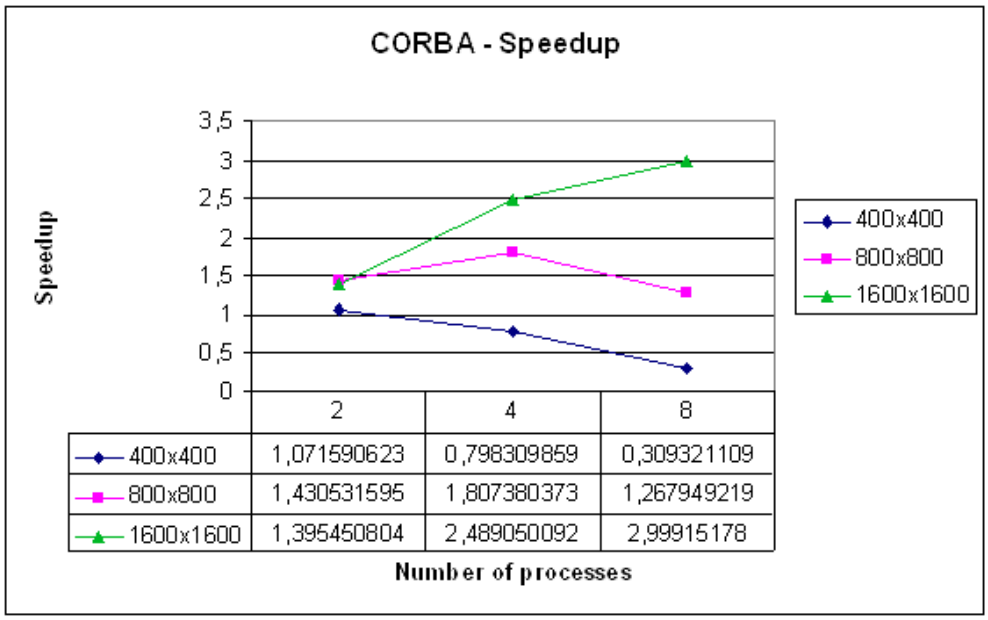
	2	4	8
400x400	47,86%	22,52%	6,44%
800x800	59,56%	43,09%	22,31%
1600x160	76,73%	54,85%	37,17%

Figure 15: HORB version – Efficiency



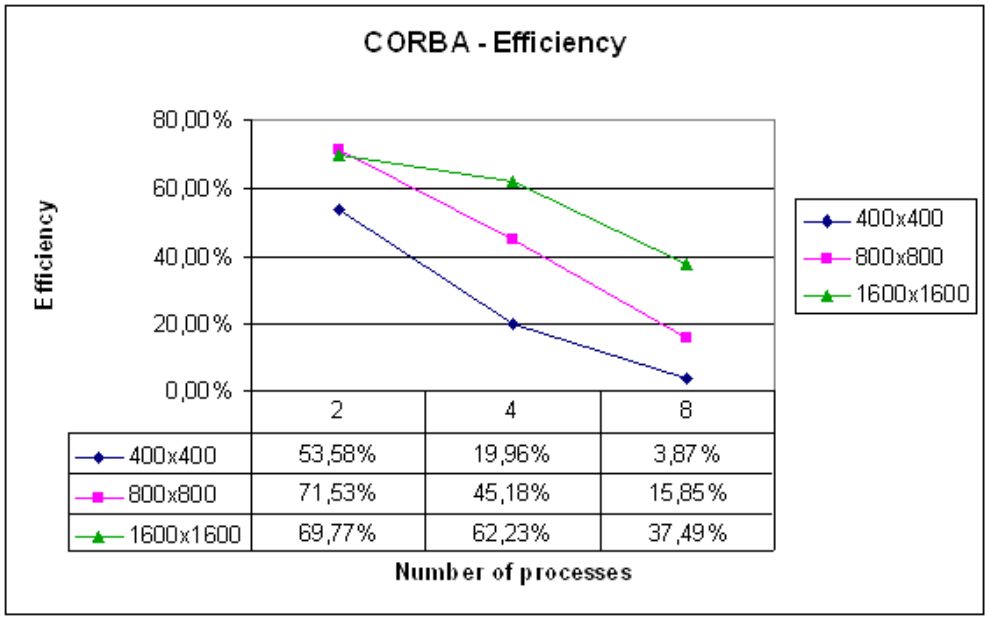
	1	2	4	8
400x400	4251	3967	5325	13743
800x800	35656	24925	19728	28121
1600x1600	289937	207773	116485	96673

Figure 16: CORBA version – Time



	2	4	8
400x400	1,071590623	0,798309859	0,309321109
800x800	1,430531595	1,807380373	1,267949219
1600x1600	1,395450804	2,489050092	2,99915178

Figure 17: CORBA version – Speedup



	2	4	8
400x400	53,58%	19,96%	3,87%
800x800	71,53%	45,18%	15,85%
1600x1600	69,77%	62,23%	37,49%

Figure 18: CORBA version – Efficiency

Interpreting Results

I note that there is a logarithmical scale on y-axe in some graphs. In all versions also the time of computation of one process was measured. In this situations the communication's or library's overhead shouldn't influence the time and really the corresponding times are nearly the same in all tested libraries, so the results can be used to compare libraries to each other.

In socket version we can see a tendency of the time to decrease (except 4-server version — it's quite interesting, but it's difficult to estimate a reason). The consequence is that the speedup tends to grow but the efficiency decline. The efficiency doesn't depend on the dimension of matrixes but only on the processes' count.

In RMI version we can see different graphs for different dimensions of matrixes. There is no speedup for matrix $400x400$. Parallelization of the algorithm has a little effect for matrixes $800x800$, but we can see stronger speedup for matrixes $1600x1600$. The graph of efficiency shows the same situation as in socket version — using two processes is the most effective.

The HORB and CORBA versions are very similar to RMI, the times differ less or more but the general tendencies are very similar.

We can see now that all tested libraries has bigger communication overhead than communication just over sockets. The bigger matrixes we calculate

the less impact this overhead has on the performance, so for bigger matrixes we can see a speedup. This speedup is not big enough so the efficiency of a parallelization is much smaller than in socket version.

There is also another aspect we must know to understand these libraries in detail. The thing is how these libraries (all ones do it in the same way) handle the incoming message (call of remote method). When an incoming message arises a new thread is automatically created to handle it. So it's possible to handle more than one message in one time. In our situation only two threads should be presented on the servers – computation and communication thread (as in socket version), since there can't be two remote calls on one server in one time in our scenario. But I have found out there are often more than two threads on the servers. Since all calls of remote methods in these libraries are synchronous it means that a garbage collector doesn't clean old communication threads quick enough and so they could affect the performance of computation.

I have found out another interesting result during testing the libraries. In first implementations I have wrapped the access to the buffer by the critical sections since both threads (communication and computation) can access it. So the computation thread needed to go through the critical section when it wanted to gain any value of pivot rows. The times of computations were very different from presented ones, for example the computation of eight-processes version on $1600 - 1600$ matrix took 755 seconds. Than I realized that communication thread access the row of the buffer only once (when it puts the pivot in the buffer) so the computation thread needn't to go through the critical section every time it needs some value. When I rewrite the program a bit the times improved significantly. It means that critical sections are very expensive expressions in Java and the programmer should use them carefully when he wants to gain the best performance.

Conclusion

As we could see the Java language has support for creating distributed/-parallel applications. If we use only socket-based communication we can gain some speedup but efficiency is decreasing. The main focus of this article was to discover the ability of some standards for creating distributed applications to be used for high performance computing. We have shown that these libraries are suitable for these purposes only for problems where their

bigger communication overhead is "melted" in the time of computation. But the speedup and also the efficiency of these libraries is much smaller against the socket version. The main domain of these standards is already in distributed applications for which the speed of response of servers is sufficient. They provide sophisticated environment that enables to create well designed applications and don't force the programmer to deal with a direct communication (that is shade off by these libraries) and so enable to create huge distributed applications.

Bibliography

[1] Doug Lea; Concurrent programming in JAVA : design principles and patterns; Addison–Wesley Logman, Inc.; 1997

[2] Sun Microsystems, Inc.; www.sun.com; java.sun.com; 2000

[3] Object Oriented Concepts, Inc.; www.ooc.com; 2000

[4] HORB specification; www.horb.org; 2000

[5] JAVA specification; java.sun.com; 2000

[6] JavaParty specification; wwwipd.ira.uka.de/JavaParty; 2000

[7] JavaNoW specification; www.plexobject.com/software/javanow/javanow.html; 2000

[8] CORBA specification; www.omg.org; 2000

Publications in the FI MU Report Series are in general accessible via WWW and anonymous FTP:

`http://www.fi.muni.cz/informatics/reports/`
`ftp ftp.fi.muni.cz (cd pub/reports)`

Copies may be also obtained by contacting:

**Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic**