ssdnm
środowiskowe
studia doktoranckie
z nauk matematycznych

Marek Nowicki

Uniwersytet M. Kopernika w Toruniu

Evaluation of the ProActive communication efficiency over the Infiniband and Gigabit Ethernet interconnects

Praca semestralna nr 2

(semestr zimowy 2011/12)

Opiekun pracy: Piotr Bała

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Evaluation of the ProActive communication efficiency over the Infiniband and Gigabit Ethernet interconnects

Marek Nowicki[1] and Piotr Bała[1,2]

[1] Faculty of Mathematics and Computer Science,
Nicolaus Copernicus University,
Chopina 12/18, 87-100 Toruń, Poland
[2] Interdisciplinary Centre for Mathematical and Computational Modelling,
University of Warsaw,
Pawińskiego 5a, 02-106 Warsaw, Poland
{faramir,bala}@mat.umk.pl

Abstract In this paper we present briefly existing parallelization tools for Java and focus on ProActive as one of the most promising approaches. We evaluate the performance of the ProActive communication on the state of the art hardware such as cluster with either gigabit and infiniband interconnect. The presented data show differences in communication efficiency for different type of Java objects. The performance data is compared with the communication based on the sockets. The obtained results validate ProActive efficiency and clearly show where communication can be improved.

1 Introduction

The parallel programming is well established activity and there are many tools for the parallelization of applications. Some tools are used for parallelization applications on the local computer and others can run distributed tasks. Because of the hardware development and availability of the multicore processors, shared memory systems, clusters as well as distributed infrastructures such as grids and clouds, parallelization of the applications is now more important than before. One should note that changes in the hardware are associated with the interest in new programming languages not being considered by traditional high performance computing. Good example is Java with its increasing performance and parallelization tools such as Java Concurrency introduced in Java 1.5 [1] and improved in 1.6 [2]. The parallelization tools available for Java does not limit to threads and include solutions based on sockets, various implementations of MPI library, distributed Java Virtual Machine, solutions based on Remote Procedure Call (RPC) and Remote Method Call (RMI).

In this paper we present briefly existing parallelization tools for Java and focus on ProActive as one of the most promising approaches. We evaluate the performance of the ProActive communication on the state of the art hardware such as cluster with either gigabit and infiniband interconnect.

2 Parallel tools for Java

Threads support is part of core Java functionality and allows for simultaneous or nearly simultaneous execution of different or the same portions of code using the same or different data. The main disadvantage is the ability to run threads only on one local machine. Libraries that support threads provide mechanisms of synchronization, the data is shared between threads. Operating system is responsible for the selection of the currently active thread. Applications using threads are usually easy to write and execute.

Java language since its beginning was designed to create threaded applications. The main object java.lang.Object, after which any other object inherits properties contains methods for notification and wait for notification from other objects. In addition, Java provides `java.lang.Thread` and `java.lang.Runnable` interfaces for creating threads and manipulate them. On the other hand, the specification does not guarantee the sequence and time selection of threads from the pool of available threads. The order depends on the implementation of the Java Virtual Machine and it may be round-robin or blind selection of threads to resume.

Another method of parallel execution of program code is to use threads and sockets (sockets). Among the advantages of the use of sockets is that the programmer has full control over the program, and he invents or adapts an existing communication protocol, so that the speed of communication can be high. On the other hand creation of such application requires a lot more work than using a ready-made solutions for communication and synchronization.

MPI (Message Passing Interface) is a standard library used to send messages between processes on different machines. It has a number of implementations in many programming languages, including C, Fortran, Python and Java. The Java implementations are usually design as interfaces to the C implementation and does not provide solutions suitable for Java programmers [3]. Low communication efficiency is another disadvantage.

Distributed Java Virtual Machines are designed to facilitate the programming and running applications on multiple computers. Their job is to run Java applications in such way that programmer does not have to be aware of that fact. The threaded application running on the single processor or any shared memory multicore system can be run on the distributed architecture without any (or almost any) changes. Despite the many years elapsed since the first attempts to create distributed JVM, there is still no satisfactory results.

Among the well-known and open application programs are: dJVM [4], JESSICA2 [5] and Terracotta [6]. The first two projects are already few years closed. The last one is still active and is dedicated to run web servers on the distributed infrastructure. Although Distributed Java VM should be able to run any Java code, in order to achieve parallel execution the user has to prepare his application in the special way supported by Terracotta JVM.

Remote Procedure Call protocol describes how to transparent procedures to run on a remote server. The server records its procedures in the register of names and waits for client notification. Then a customer reports a desire to execute the

procedure and sends the input data (arguments). After processing the request, the server responds to the client or the result is an error information.

Remote method call is a protocol similar to RPC created for the Java language. The main difference is that it uses an object model instead of procedure. Another difference is the shift away from rigid client-server architecture. In this case, it is possible to use a distributed architecture. The RMI allows object whose method is invoked, to be on a different JVM, which may be located on another computer. Remote object is used almost identically to a local one. The RMI heavily uses serialization for communication and this appears to depreciate efficiency. Dedicated version such as KaRMI have been developed but their utilization is limited [7].

ProActive [8] is a Java library used to parallelization of applications using distributed and multithreaded models. With this library, it is easier to program and run applications on multicore processors, clusters and Grid infrastructure. In contrast to the distributed Java virtual machine, ProActive does not require any modifications to the JVM, so that runs on any operating system that contains the implementation of the JVM. Below we describe ProActive in details.

3 ProActive

The main principles of ProActive is that calculations are scattered over availiable nodes. Objects which can be accessed remotely are so-called active objects and extend ActiveObject class. Results of calls of the methods from the active objects are stored in FutureObject. Communication and computation is run, if possible, asynchronously, the results of the remote calls are transferred only when required (a mechanism wait-by-necessity).

Active object can be created from almost any type of class - it must be public and extensible (the class definition must not contain the `final` modifier) and have constructor with no arguments (though other constructors may exist). Active objects have their own thread responsible for calling public methods. The creation of this thread is done by ProActive and the programmer does not need to implement it, unless he wants to change the order of requests handling . Active objects can be created on any computer involved in the calculations. Location of the active object is selected by ProActive, but the developer, if desired, may have influence it. Using the active object is transparent - created the active object is projected on the type of class, from which arose, so that communication takes place via calling public methods of the object. Each call creates a message containing the parameters that are serialized Java objects.

In most cases, after calling the asynchronous method on the active object the future object is returned instantly. It contains a temporary result of the calculations, to be completed when the asynchronous method is finished. If future object is used before it is filled, the thread is stopped until the results will be available. If the method declares that it does not return any result (void) and do not declare throwing an exception, the call is asynchronous. However, if the return type is a simple type (eg `boolean`, `byte`, `int`, `double`) or the final class

(eg, `java.lang.String`), or declares the method can throw an exception, it is not possible to perform asynchronous method call.

`Active Object` and `Future Object` create additional components during sending requests and receiving responses. Stub component is the mediator between the class created by the programmer and the proxy component. Proxy component allows for asynchronous method calls - sends a query to a remote node containing Active Object. In addition, Active Object stored in the request queue contains content created by a programmer, that is the logic performed on a remote node.

ProActive calls allows for point - point as well as group communication. The group consists of a set of Active Objects. Each group member can send requests to all members of the group and can receive responses from all group members. The result of group communication is also a group, each response is a Future Object.

In ProActive there is available mechanism for synchronization: one can wait for replies from all Active Objects, wait for the N-th response or a response from the given Active Object.

The ProActive nodes are divided into: physical and virtual nodes. Physical node is located on the JVM. Running it on a remote machine requires the use of an appropriate protocol (eg. rsh, ssh, login, GLOBUS, UNICORE). Physical node creates and provides access to the Active Objects.

Virtual Node in ProActive is an abstract concept which allows to separate execution from the actual physical infrastructure. Virtual Node allows for flexible application deployment, rapid replacement of computers performing calculations and elimination of the details of configuration (host names, used protocols, ports and network interfaces). Virtual Node is defined in the configuration file and is identified by a unique name.

Example scenario of the calculations using ProActive is as follows:

1. ProActive creates the RMI registry.
2. ProActive deploys Active Objects on the nodes.
3. Active Objects method call creates a stub, which communicates with the proxy at the appropriate node.
4. The Future Object is returned.
5. The program continues to run.
6. On the node the queue of requests is checked timely and request is being processed.
7. After the request is processed, the result is returned to the calling program and stored in the returned Future Object.

4 Efficiency of ProActive communication

The usage of ProActive for application parallelization is known, however our early experiments showed large memory and communication overheads. The problems became important for large scientific applications with the complicated

objects marked as ProActive Active Objects. This lead us to perform detailed studies of the communication efficiency of the ProActive.

We have decided to use the default RMI protocol using state of the art hardware. In particular tests have been run on the cluster built of 64 bit AMD Opteron Processors 2435. Each processor is six cores at 2600 MHz with 512 KB cache, 32 MB RAM running 64-bit Java virtual machine, Oracle version 1.6.0_23. The nodes are connected with the Gigabit Ethernet and Infiniband cards.

Initially, tests were performed using the ProActive version 4.3.1, but during testing, a new version of the ProActive library 5.0.0 became available and tests has been performed using a new version.

4.1 Point - point communication

The tests consisted of a combination of two different nodes selected randomly. The different data structures has been transmitting between nodes:

- array of integers: `int[]`
- array of long integers: `long[]`
- string: `String`
- array of booleans: `ArrayList<Boolean>`
- vector of booleans: `Vector<Boolean>`

Number of elements in the array and the collection has changed from 10^7 to 10^{10}.

In order to compare the speed of sending data depending on the used data types, as well as to compare the speed achieved on different types of communication the results have been normalized by counting the number of bits needed to store the data in uncompressed form. We have assumed that a byte contains exactly 8 bits.

	Serialization		Conversion		Garb.Collect.(x86)		Garb.Collect.(x64)	
Type	header	element	header	element	header	element	header	element
`int[]`	27	4	16	4	16	4	24	4
`long[]`	27	8	16	8	16	8	24	8
`boolean[]`	27	1	16	1	16	1	24	1
`ArrayList<Integer>`	125	10	40	20	40	20	64	32
`ArrayList<Long>`	122	14	40	20	40	20	64	32
`ArrayList<Boolean>`	98	5	72	4	40	4	64	8
`Vector<Integer>`	224	10	40	20	40	20	64	32
`Vector<Long>`	221	14	40	20	40	20	64	32
`Vector<Boolean>`	197	5	72	4	48	4	64	8
`String (UTF-8)`	13	1	40	2	48	2	64	2

Table 1. Size of the header and a size of a single element calculated using various methods.

We have used three mechanisms to calculate the size of the data:

- the number of bytes needed for serialization of the object
- recursive conversion of the volume occupied by individual components of the object
- the difference in the amount of free memory before and after the data initialization using `GarbageCollector`, to assess the amount of memory reserved.

As shown in the Table 1. the usage of the GarbageCollector to calculate the correct memory size has failed. Calculated memory size is ambiguous – size is different for 32-bit (x86) and 64-bit (x64) version of Java Virtual Machine. Other mechanisms results not differ in case of different version of JVM. In the case of the conversion and serialization calculated data size differs, due to the different characteristics of each. Serialization needs more data to be stored in the headers of the data (eg. by writing the names of types of stored variables), however in some cases allows for reduction of the size of the data element.

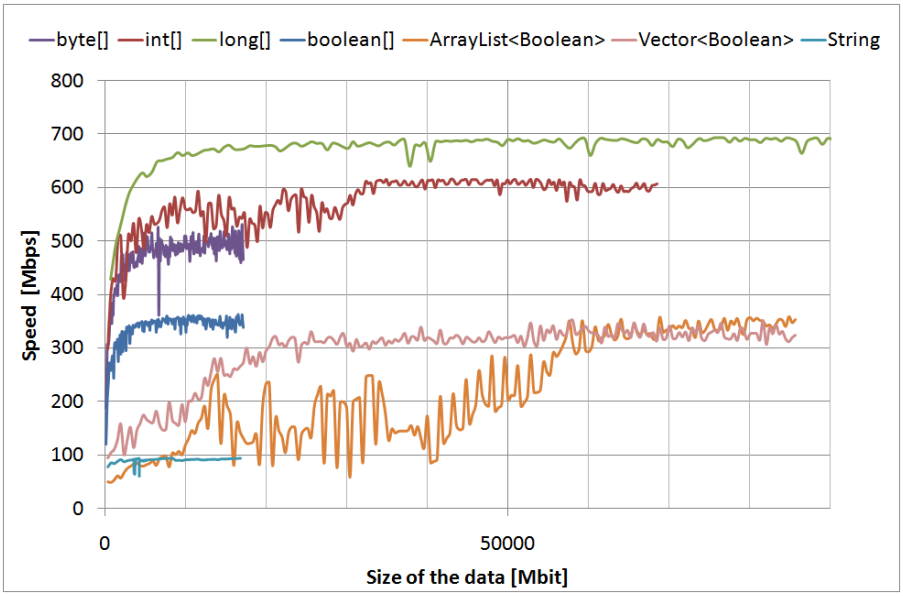

Fig. 1. Data transfer speed for different communicated objects.

In the Figure 1. the data transfer speed for different types of the communicated object is presented. With the size of data the communication speed increases until it reaches saturation. The obtained maximal speed is determined by the available bandwidth and efficiency of operations performed by the ProActive library during the preparation of the data for the transfer as well as by the necessary postprocessing.

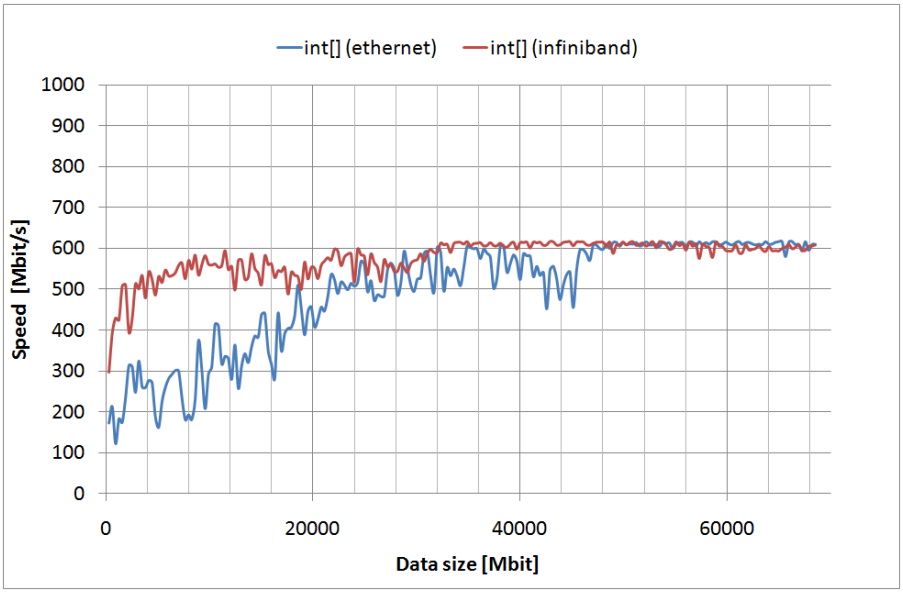

Fig. 2. Data transfer speed for ethernet and infiniband transport layers.

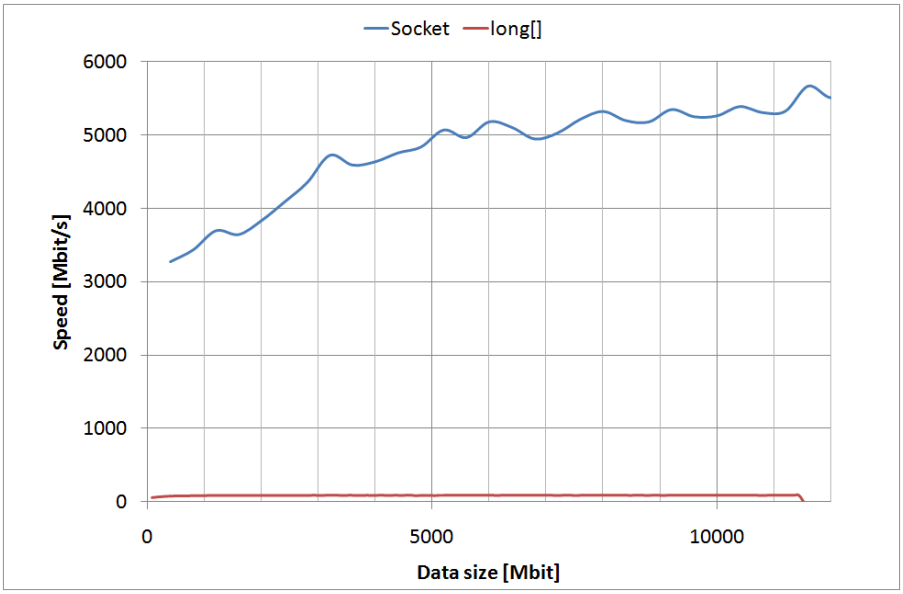

Fig. 3. Data transfer speed for ProActive using RMI and Java sockets.

Depends of the type the data transfer speed varies significantly, from ca. 100 MB/s for `String` to 700 MB/s for `long[]` data structure. As one could expect the data transfer for more complex data structures such as arrays and lists is lower than for simple data structures such as indexed variables. The data transfer speed for the String object occurs to be the lowest one. Quite low is data transfer speed for the array of boolean variables.

As presented in the Figure 2., the communication speed does not depends on the type of the communication channel. The maximal reached data transfer speed for the gigabit ethernet and infiniband communication channels are the same. The infiniband can be characterized by lower latency which allows to reach maximal transfer earlier. The data transfer is significantly lower than available bandwidth, which is 1000 Mb/s for gigabit ethernet and 20 000 Mb/s for infiniband. This observation is confirmed by the comparison of data transfer speed for ProActive and Java sockets presented in the Figure 3. In both cases infiniband has been used and data transfer obtained with the Java sockets is almost 10 times larger than in the case of ProActive.

The communication speed has been compared across two recent ProActive versions, namely 4.3.1 and 5.0.0. The data presented in the Figure 4. shows that there are some improvements implemented in recent version of ProActive. The differences are small for array of integers, but are quite significant for array of boolean variables `ArrayList<Boolean>`. This effect confirms, that communication efficiency depends mainly on the ProActive library.

4.2 All to all communication

The another type of communication pattern important for the efficiency of parallel applications is synchronization or all to all communication. The performance data has been generated by sending of single boolean variable from given node to all other nodes. In return the modified variable has been received. The total time for such operation has been recorded and presented in the Figure 5 and Figure 6. The results are obtained using two different modes implemented in the Proactive: with and without scatter.

The results show good scaling for the communication model using scattering and significantly worse while scattering is not used. The presented data shows importance of the implementation of the global communication algorithm and its influence on the scaling. The performance date has been collected on the multicore system which allowed us to run multiple virtual nodes on the single physical one. While multiple virtual nodes share the same physical node the communication tends to be slower which reflect usage of one RMI process for the physical node. This difference is higher when scattering mechanism is not used.

5 Conclusions

ProActive library allows for almost seamless parallel programming in Java. All data transfer operations as well as synchronization events are performed auto-

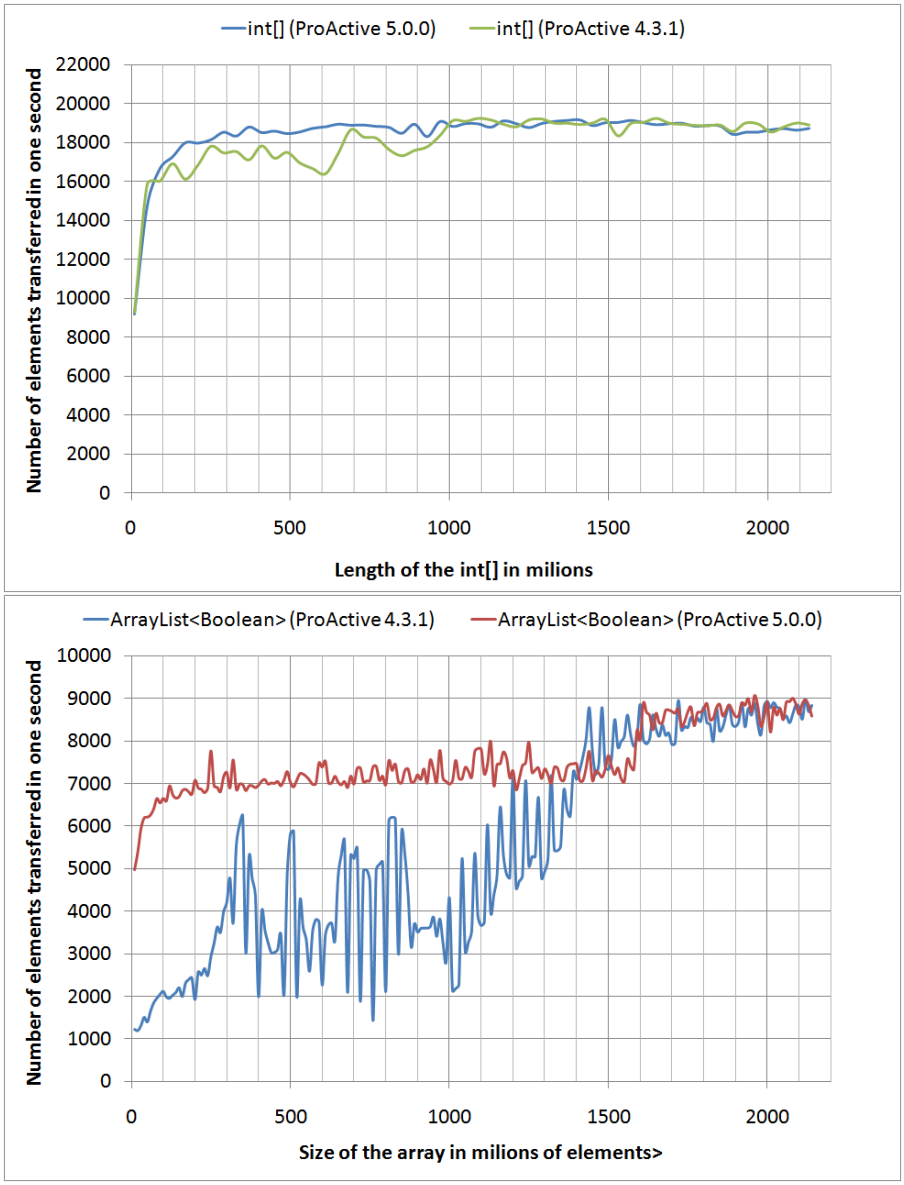

Fig. 4. Number of array elements transferred in one second for different data sizes for ProActive versions 4.3.1 and 5.0.0. The data is presented for array of integers `int[]` (top) and array of booleans `ArrayList<Boolean>` (bottom).

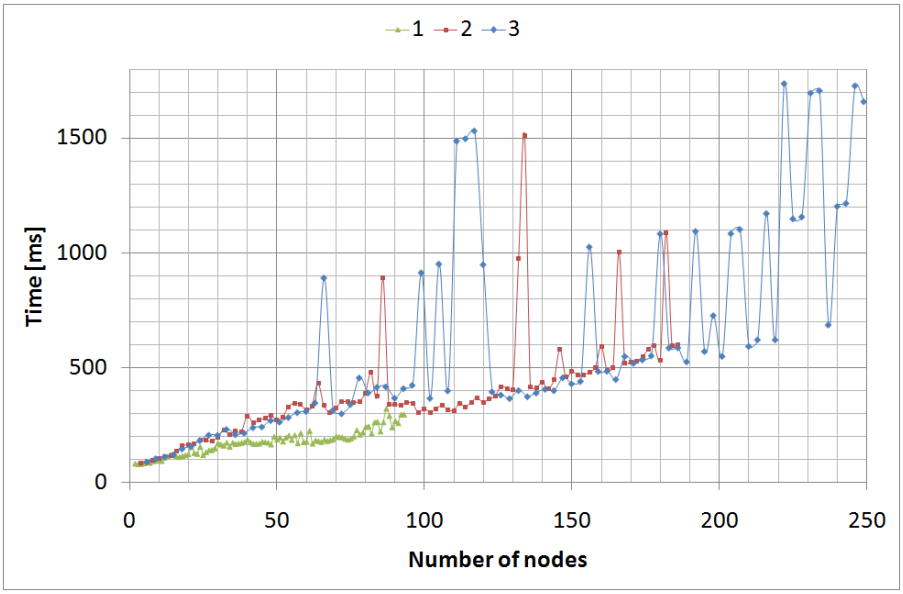

Fig. 5. Time of ProActive all to all communication using scattering. The results for different number of ProActive virtual nodes located on the physical node is shown.

matically and no special programming constructs are needed. Running of the ProActive application on the distributed infrastructure requires only creation of the configuration files with the description of the nodes used for calculations. However, the communication efficiency significantly depends on the type of data transferred. Moreover, the results are far from achievable data transfer speed available using for example `java.net.Socket` and practically does not depend on the communication layer (ethernet or infiniband). Obtained results show that communication overhead introduced by the ProActive, and in general by the object oriented approach, is significant and can cause fproblems. The main reason of slow communication is poor efficiency of the serialization of the communicated object. The another problem is high memory requirement of ProActive coming from the multiple copies of the transferred objects stored in the JVM. This problem is especially important when complicated objects are marked as `Active Objects` or `Future Objects`.

Acknowledgements

This work has been performed using PL-Grid infrastructure.

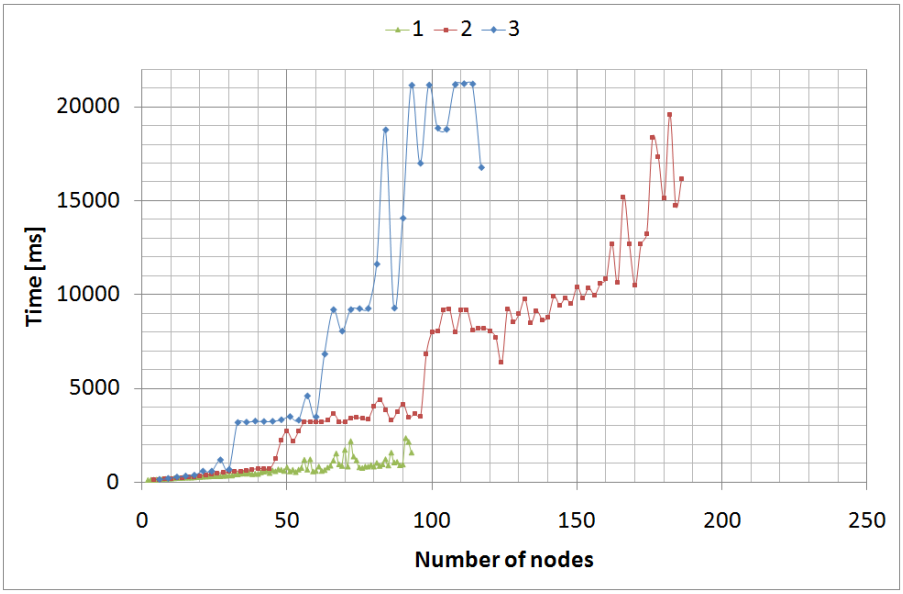

Fig. 6. Time of ProActive all to all communication without scattering. The results for different number of ProActive virtual nodes located on the physical node is shown.

References

1. Java 2 Platform Standard Edition 5.0, New Features and Enhancements http://docs.oracle.com/javase/1.5.0/docs/relnotes/features.html
2. Java Platform, Standard Edition 6, Features and Enhancements http://www.oracle.com/technetwork/java/javase/features-141434.html
3. Carpenter B., Getov V., Judd G., Tony Skjellum and Geoffrey Fox. MPJ: MPI-like Message Passing for Java. Concurrency: Practice and Experience, Volume 12, Number 11. September 2000
4. Wenzhang Zhu, Cho-Li Wang, Francis C. M. Lau.: Lightweight Transparent Java Thread Migration for Distributed JVM, Parallel Processing, International Conference on, p. 465, 2003 International Conference on Parallel Processing (ICPP'03), 2003
5. Wenzhang Zhu, Cho-Li Wang, Lau, F.C.M.: JESSICA2: a distributed Java Virtual Machine with transparent thread migration support, 2002. Proceedings. 2002 IEEE International Conference on Cluster Computing, pp. 381- 388, 2002. DOI: 10.1109/CLUSTR.2002.1137770
6. Boner J. and E. Kuleshov E.: Clustering the Java virtual machine using aspect-oriented programming. In AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development, 2007.
7. Nester Ch., Philippsen M., and Haumacher B.: A more efficient RMI for Java. In Proceedings of the ACM 1999 conference on Java Grande (JAVA '99). ACM, New York, NY, USA, 152-159. 1999 DOI=10.1145/304065.304117

8. Baduel L., Baude F., and Caromel D.: Efficient, flexible, and typed group communications in Java. In Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande (JGI '02). ACM, New York, NY, USA, 2001 p. 28-36. DOI=10.1145/583810.583814 http://doi.acm.org/10.1145/583810.583814