DPJ: JAVA CLASS LIBRARY FOR DEVELOPMENT OF DATA-PARALLEL PROGRAMS

V. Ivannikov, S. Gaissaryan, M. Dommrachev, V. Etch, N. Shtaltovnaya

Abstract

Problem of Java language usage of data-parallel programs using the SPMD model of parallel execution is disscused in the paper. The sequential components of parallel program are executed in parallel on distinct JavaVMs running on processors of the parallel computer. Links between the parallel program components are carried out by means of the standart message passing interface package MPI. Parallel extension of Java is made by means of Java itself, namely by DPJ class library, containing the set of Java classes and interfaces. Description of the DPJ library is a main subject of this paper. The scope of the paper is restricted to implementation of the DPJ library for set of JavaVM .

1 *Introduction*

The problem of usage of the **Java** lenguage [1] for data-parallel programs development within the framework of *SPMD (Single Program, Multiple Data)* model [2] is discussed in the paper: the parallel program represents a set of sequential functionally identical components. Each component represents the **Java**-program executed on separate **JavaVM**. All components and their **JavaVMs** work in parallel on processors of the parallel computer[1], so that on each processor appropriate **JavaVM** interprets one component of the parallel program. Thus, being a **Java**-program each component may use parallel capabilities of JavaMV (i.e. to be executed in several threads [1]).

We add new parallel capabilities to **Java** by means of **Java** itself: we implement **DPJ** library, containing the set of **Java** classes and interfaces providing these facilities. Describtion of **DPJ** library is the main subject of the paper.

Nowadays the activity on organization of parallel computations using **Java** language is of great interest for teams working on development of various parallel systems. It is possible to say so considering the regular publications on this subject in the Internet [3]. It is worth to mention two

[1]Term the parallel computer means either multiprocessor supercomputer with distributed memory or local (generally sreaking, inhomogeneous) network of computers.

approaches to solution of a problem of parallel computation organization using **Java** lenguage:

• Development of **Java**-interfaces to parallel programs written using other parallel progamming languages: HPF [4], HPC [5], pC++ [6], MPC++ [7], etc.;

• Development of special facilities for design of parallel programs using the **Java** language [8]. The former approach only conventionally could be treated as parallel programming using **Java** language, because only the calls to the parallel library functions are programmed on **Java**, but the parallel programming of these function bodies is carried out on other parallel languages. The use of parallel libraries can provide high enough efficiency, and also allows one to use the legacy highly-efficient parallel programs. The disadvantages of this approach are in restricted compatibility of such libraries due to the problems of recompilation and installation of runtime systems of these libraries to another platform. Additionally it is difficult to control the library efficiency in case of porting it to another platform as these libraries usually work effectively only on parallel computers for which they were designed. At last, the possibilities of distribution of such libraries are limited as they require usage of expensive compilers and runtime systems.

We believe the second aproach is preferable because only this approach supports the full cycle of development of the parallel **Java**-program [9]. The standard communication package **MPI***(Message Passing Interface)* [10] has been implemented for most of parallel plaforms and is used to support communications among components of the parallel program. The parallel program entirely developed on **Java** can be executed on any set of **JavaVMs** working in parallel and interacting via **MPI** functions. It may be easy debugged and modified due to reflexive properties of **Java** language. Certainly, the performance of the parallel **Java** -program in interpetation mode concedes the performance of compiled programs developed on other parallel languages such as `pC++`,`MPC++`, etc. However, the distribution of the parallel **Java**-program bytecode allows to use it immediately and preliminary evaluate performance improvement of the application caused by use of this parallel **Java** -program. Thus there is no need in the complex and expensive software installations as with parallel programming systems described in first approach.

Only after the fitness of the parallel **Java**-program is proved the problem of its performance increase will arise. We propose three ways to solve this problem. The first is similar to the first approach to **Java** parallel programming: instead of bytecode the machine code obtained as a result of compilation of `C++` program which is functionally similar to given **Java**-program and has the same **Java** -interface as source **Java**-program. The second way is to use the **Java** compiler and/or **JavaVM** bytecode compiler

directly to machine codes (**Java**-to-native compilers) [11,12], optimizing converters of **Java** -programs [13], etc. This allows to increase performance of the parallel **Java**-program components and to draw it near the performance of programs on other programming languages (for example, `C/C++`,`FORTRAN`, etc.). The third way proposes the development of the optimizing parallelizing **Java** compiler directly to machine codes of parallel computers (Parallel **Java**-to-native compilers).

The parallel programing system **DPJ** described in the paper suggests the following tecgnique of parallel **Java**-programs development:

1. The parallel program is developed on **Java** language with the aid of special **DPJ** library classes described below;

2. Then the program is compiled to bytecode and dabugged on a parallel computer in interpretation mode;

3. The evaluation of the program parallel properties is performed (in particular the degree of the program acceleration under parallelization, program scalability, etc. are evaluated);

4. If the program reveals acceptable outcomes of the parallelization degree, one of three described above ways to performance improvement is applied;

5. The further performance enhancement will involve the replacement of the **MPI** package to the specially developed optimized run-time support system.

In this paper the presentation is limited to implementation of the **DPJ** library for set of **JavaVMs**. The implementation of mentioned ways of performance improvement will be covered in the succeeding publications.

It is nessesary to note that alongside with implementation of the **Java** classes library there is an implementation of the `C++` programming system (**DPS**++) [14] using the `C++` templates. The **DPS**++ library is compatible with the standard STL library [15] included into the `C++` standard [16]. The **DPS**++ implementation will provide high enough efficiency for parallel programs designed under this library.

The **DPJ** parallel capabilities include the main components of the library: networks and subnets (subnetworks), distributed containers and their iteratos and parallel algorithms. Additionally some aspects of the **DPJ** implementation using the **MPI** package, as well as distinction between parallelism provided by **DPJ** library, and built-in **Java** thread parallelism [1] are considered in the section 2.

The section 3 is devoted to the detailed description of networks and subnets definition facilities. In the current DPH implementation the definition of all networks and their subnets is performed dynamically with the aid of the **MPI** package.

The definition facilities and basic types of distributed containers are described in section 4. For effective processing each type of distributed container assumes a subnet topology it is distributed on to be defined. The information about the subnet topology is passed to the **MPI** package for more effective mapping of the virtual network onto real one.

The types and definition facilities of iterators of distributed containers are described in section 5. The distributed container iterator provides a simultaneous access to all elements of one subset of nodes of this container. Depending on a sort of subsers there are following types of iterators: unary (contains single node subsets), overall (contains subset of all nodes), and multiple (contains arbitrary subsets of nodes).

The section 6 contains the brief description of the parallel algorithms implemented in the current **DPJ** version used most frequently when developing parallel programs. The parallel algorithm is parallel program whith ensures effective execution of standard operations on any types of distributed containers.

The usage of all **DPJ** basic components is described in section 7 as an example of parallel program.

2 *The Brief Description of Parallel Capabilities of the DPJ Library*

As has been mentioned above we treat parallel programs working under SPMD model. Such program is a set of functionally identical components consisting of either bytecode or machine codes of concrete paralel computer. In the first case component represents the **Java**-program executed on separate **JavaVM**; and all components and their **JavaVMs** work in parallel on processors of the parallel computer, so appropriate **JavaVM** internets one component of the parallel program on each processor. In the second case each component represents a machine code image of a **Java** -program, and functionally identical components in machine codes are executed in parallel on processors of the parallel computer. From now on the component of the parallel program will mean the bytecode implementation of this component.

MPI package is used for organization of communiction and data exchange between components. Providing a set of primitives to handle the

message passing **MPI** package allows to hide a physical structure of computer processors, used network hardware, as well as operating systems peculiarities. One of important primitives of the **MPI** package is *process*. The process is a unit of the program execution, and the message passing is performed only among processes. The **MPI** package allows to mapp to one component of the parallel **Java**-program. Thus the parallel **Java**-progran is mapped onto a set of **MPI** processes. It is convenient to refer this set as a *network*, each process being a *node of this network*. The network is conceptually created by a special starter class thet defines the power of network and the parallel program starting parameters.

Using the model described a component of the parallel program could be associated with a node it is executed on. One could use parallelization facilities defined in the **Java** language environment (classes java.lang.Thread, java.lang.ThreadGroup) while every node executes the **Java**-program. These classes use a shared memory model parallelism. In the paper only the distributed memory parallelism model is discussed, i.e. we suppose that each component of the parallel program runs in the separate address space, and threads run within one node achieving the real parallelism on a node with complex structure (e.g. the **MPI** process is executed on SMP architecture which supports thread distribution among different processors). The main goal of the research is development of parallel execution control, as well as facilities for distribution and axchange of data among nodes.

One of the program parallelization techniques is partitioning the program into independent parts. Let, for example, the program consists of two independent parts A and B, which could be executed in parallel. It is convenient to define a subnet α of the network for execution of part A of the program and subnet β for execution of part B. Thus both parts A and B of the program are executed only on an appropriate subnet, and does not executed of another subnet[2]. In turn, one may define subnets of every subnet. The nodes of the network of subnet are enumerated by integers from 0 up to $n-1$, where n is the power of the set of network (or subnet) nodes.

As in any other object-oriented language in the **Java** language it is convenient to represent the sets of similar objects by container objects (*containers*) [17]. Examples of containers are arrays, vectors, lists, set, key set, atc. Each object that is accesed through the container is *element of this*

[2]In spite that in the program α and β subnets are defined, according to SPMD model the same code is located on every node, i.e. there is a code of both A and B parts on each node. Nodes of subnet α will never execute the code of part B, and nodes of subnet β will never execute the code of part A. The execution of one or another part on the node is performed by a switch which determines which subnet owns this node and initiates execution of an appropriate part of the program.

container. The container *iterator* is an object which provides access to the elements of this container. In the sequential program at any given moment of time iterator provides access at most to the one element of container. It is convenient to use iterator in the loops processing the container elements.

We introduce implementation of data distribution using distributed container, which arranges its elements so that each node of container owns one element which is placed on the corresponding node of container subnet N. The subnet N is specified when the distributed container is instantiated.

As in usual container, each element of distributed container can have references to other elements of this container: in this case the reference specifies a node where the addressed by this reference element resides. Under distributed container reduction operations may be defined which are executed involving all elements of the container in parallel with a peak afficiency. There are following reduction operations: counting container elements, obtaining element index, summation of all container elements (for example, distributed array), detecting maximum and minimum, etc.

Access to distributed container elements is provided by container iterator. The distributed container may have an arbitrary amount of iterators.

The distributed container iterator is object distributed across the same subnet as its container and providing simultaneous access to all elements of one sebset of container nodes. This subnet of container nodes is called *iterator value*. The set of all iterator values forms a covering of the set of container nodes (with or without intersections). Two operations are defined on the iterator: assignment and reassignment of its value. With the aid of these operations iterator can in turn accept all its values wthich altogether form a covering of set of all container nodes.In particular, one may define the iterator providing simultaneous access to all elements of the distributed container at once (overall iterator), iterator defining simultaneous access only to the set of one element of the container (unary iterator), ets. Most frequently used iterators for each of distributed container types are available in the **DPJ** class library.

Distributed container element could be an object implementing the *parallel algorithm interface*. Such an object has a method run() defined by the user which represents the body of a component of parallel algorithm that is executed on every node. The execution of parallel algorithm begins by the call of method start() on all nodes of the subnet which the appropriate object is distributed on. Iterator of such container returns current subset of nodes executinng method run(). Examples of parallel algorithms are parallel search, parallel sorting, etc.

At run time there is a need of date exchange with external sources, e.g. data should be read from or written to a disk file. A special subsystem of parallel input/output is supposed to be developed to meet these needs.

Currently this problem is solved by means of standard **Java** facilities (e.g. input/output can be performed by the single node to its file).

In our implementation of this concept using **Java** all described system objects, namely subnets, distributed containers, their iterators, and the parallel algorithms are represented by the hierarchy of library and user classes and interfaces. Their semantics and syntax are described in the following sections of this paper.

3 *Subnet Definition*

The definition of subnet of some network (or subnet) is performed by instentiation of the special Subnet class. Each subnet has the parent network (or subnet) and can have an arbitrary amount of the child subnets. The brief description of the Subnet class methods is given below:

- Define a subnet
- Get amount of subnet nodes
- Operators on subnets:
  - Union
  - Intersection
  - Complement
- Get space of this subnet free from its child subnets (complement of all child subnets union)
- Get a subnet of nodes included in the minimal number of child subnets

The Subnet class object representing the network is instantiated when the parallel program is loaded. This object has no parent network.

The Subnet class contains fields and methods designed for a load balancing control. When a subnet A of the network N is instantiated, the values of loading counters of the nodes of network N included in A are increased by one. Methods getFree return array of numbers of nodes which have zero values of loading counters. The method getMinimumLoaded returns array of numbers of nodes with the minimal values of loading counteds.

In current implementation of the library definition of the network (or subnet) causes creation of appropriate **MPI**grour and communicator. The **MPI** communicator creation requires one act of collective communication on appropriate parent net.

4 *Distributed Containers*

The *distribured container* locates its elements one per node of a subnet which is specified in its definition. The distributed container can be instance

of any class which implements the appropriate interface. The distributed container elements are **Java** object instances. Primitive types should be represented as instances of appropriate objects (e.g., Character, Integer, etc.) before placing them into the container. The distributed container elements may occupy only a part of nodes of subnet which the container is distributed on, and the number of elements may vary during execution of the program. The initial amount of nodes is srecified when the container is instantiated. When node is added to distributed container the situation when all nodes of a subnet are already occuried by other elements of the container may occur. In this case an exception occurs.

The access to elements of distributed container is provided by its iterators. Strong typing of the container elements can be performed with the aid of typed iterators and adapters. The interfaces hierarchy of the **DPJ** distributed containers is given on figure 1.

Fig.1. Interfaces and library class hierarchy of distributed containers.

The distributed container interfaces are derived from the root DContainer (distributed container) interface. This interface defines the following methods:

- Definition of the container on subnet.
- Obtaining the subnet of the container.
- Obtaining current (or maximum) amount of nodes.
- Obtaining (or setting up) the container element value on local node.
- Scattering of values from the one-dimensional **Java**-array to be the distributed container element values (scatter). The number of elements in the **Java**-array should correspond to the container size. The distributed container element located on a node with an index i will be assigned the value of the **Java**-array element with index i.
- Gathering of values of the distributed container elements into **Java**-array (gather). The order of elements in the array after operation corresponds to the order of node indices in the container.

Alongside with interface definition **DPJ** contains appropriate library classes implementing these interfaces (for example, the interface DArray is implemented by the DArrayClass class). Any pair of the container nodes may be *linked*, one of these nodes being called parent node, and another one - a child node. The link between nodes of container means that nodes should be topologically close to each other, because the intensive data exchange may take place between them. Link is the only information about the real network topology, that can be used when mapping the virtual network on the real physical one. **DPJ** library distributed containers (*standard distributed containers*) have pre-defined links between their nodes. When a new node is added to the container its link with other nodes are established in a standard way described for each type of library container classes.

DPJ user can either use standard distributed containers or define his own ones implementing the DUserContainer interface (or interface derived from it). This interface inherits DContainer interface and contain methods for definition of links between container nodes. The use of standard distributed containers, as a rule, ensures higher efficiency, then that of user defined ones.

The following distributed containers are included in the **DPJ** library:

- *Distributed array* is a distributed container with constant amount of elements (nodes) which have not links among them. Random access iterators are applied to distributed arrays. The amount of nodes in the array is specified when array is specified when array is defined and remains constant during program execution.
- *Distributed vector* is similar to the distributed array except the amount of nodes in the container an during execution of the program.
- *Distributed list* is a distributed container for which a head node, inter-

mediate nodes and tail node are defined. If the list consists of one node, this node is head node and tail node at once. Sequentail access iterators are applied to distributed lists.The amount of elements in the list can vary during program execution. The distributed list interface defines the following methods:

* Add (remove) a list node.
* Obtain iterator consisting from a hear (or tail) node of the list.
* "Shift" the sublists specified by iterator value i: in each sublist the element value on node $k+1$ becomes equal to the value from node k; the element of the first node of the sublist saves its value.

• *Distributed ring* is similar to the distributed list except its head node is a child of tail node.

• *Distributed tree* is a distributed container, which has a root node, nonterminal nodes and leaf nodes.If the tree consists of one node, this node is root node and leaf node at once, and it has neither child nor parent nodes. In other cases the root node has no parent node, but has some child nodes.The leaf node has no child nodes but has one parent node. All nonterminal nodes have one parent node and some child nodes and these nodes are distinct. Sequential access iterators are applied to distributed trees. The amount of elements in a tree can vary during execution of the program. The distributed tree interface defines the following methods:

* Add a new level of a tree nodes. The nodes of an added level become leaf nodes, and former leaf nodes - nonterminal ones.
* Add new child node.
* Get iterator providing access to the root node of the tree.
* Get iterator providing access to the leaf node of the tree.

5 *Dictributed Container Iterator*

Distributed container iterator is an object distributed on the same subnet as its container and providing simultaneous access to all elements of a subnet of nodes of this container. This container node subset is called *iterator value*. The set of all iterator values forms a covering of the container node with or without intersections).Two operations are defined for the iterator: assignment and reassignment of its value. With the aid of these operations iterator can turn accept all its values which altogether form a covering of the set of all container nodes. Instance of any class which implement the appropriate interface can be iterator of distributed container.

• *Unary iterator* (DUnaryIterator) defines an access only to the single container node simultaneously.

• *Multiple iterator* (DMultiIterator)defines an access to subset of the container nodes simultaneously. The power of this subset can vary from value to value.
• *Overall iterator* (DAllIterator) defines an access to all container nodes simultaneously.

In implementation on **Java** the distributed containers iterators library classes are implemented with the aid of interfaces given in table 1.

	Unary interface DUnaryIterator	**Multiple interface** DMultiIterator
Input Iterator	`Unary Input Iterator`	`Multi Input Iterator`
Output Iterator	`Unary Output Iterator`	`Multi Output Iterator`
Forward Iterator	`Unary Forward Iterator`	`Multi Forward Iterator`
Bidirectional Iterator	`Unary Bidirectional Iterator`	`Multi Bidirectional Iterator`
Random Access Iterator	`Unary Random Access Iterator`	`Multi Random Access Iterator`

Overall Input Iterator	Overall Output Iterator
DAllInputIterator	DAllOutputIterator

The notes:
1. In the table the bold font marks iterator interfaces,
normal font - library classes
2. Overall iterators provide simultaneous access to all elements of the distributed container.

6 *Parallel Algorithms*

DPJ library contains standard classes implementing frequently used parallel algorithms for processing the distributed container elements. All of the library *parallel algorithms* implement the interface ParallelAlgorithm containing the following methods:
• Start (stop or suspend) method run(), runTop(), or runBottom().
• Methods run(), runTop(), or runBottom() themselves. Currenyly in the library implemented following parallel algorithms:
• Applying - applies the specified functional object to the container elements.
• Coping - copies the distributed container (possibly, reverting the element order).
• Comparing - parallel comparing of elements of two distributed container.
• Finding - parallel search of elements in the distributed container using the template.

- Filtering - parallel selection of elements in the distriuted container using the template.
- Replacing - parallel replacing of the distributed container elements.
- Sorting - parallel sorting of the distributed container elements.
- Transforming - parallel transforming of the distributed container.

7 *Structure of the Parallel Program*

The parallel program is represented by the **Java** class containing the method main which begins the program execution. The method main of the parallel program is executed on the network, which is specified when program is loaded. This network is available through the static netWorld object of the Subnet class which is created when this class is loaded. Using methods of the netWorld object it is possible to obtain amount of **JavaVM**s the parallel program is running on.

Definition of subnets of the netWorld should be done for parallel execution of different parts of parallel program. For example, let the program consists of two independent parts A and B, which can be executed in parallel. One may define subnet α (for execution of the part A) and subnet β (for execution of the part B) as subnets of netWorld. The patrs A and B are represented by distributed container classes A_class andB_class implementing intrerfaces DContainer (or interfaces derived from it) and ParallelAlgorithm (of interfaces derived from it). The code of parts A and B of the program ahould contain methods run() of the classes A_class and B_class overriding appropriate methods ParallelAlgorithm interface. When odjects A_object and B_object of classes A_class and B_class accordingly are instantiated the appropriate subnets α and β which the parts A and B are distributed on should be specified as the parameters of methods distribute(). The start of each part is performed from the function main by method start() of the ParallelAlgorithm interface: statement A.start() starts the execution of the part A and B.start() starts the execution of the part B.

If the elements of the distributed container are indpendent then methods run() can be started in parallel for all these elements providing the maximum possible parallelism. In this case it is possible to use overall iterator.

If there are dependencies among the distributed container elements then methods run() can be started in parallel only for a subset of the distributed container elements. Each such subset should be specified by the distinct iterator value. Thus, correctly selected iterators help to parallelize the program when the data dependencies are present.

The use of the **DPJ** library for development of parallel **Java**-programs is shown by the following example:

```
package examples;
```
Program from the examples package: parallel sorting using *QuickSort* parallel algorithm [2].

```
import ru.ispras.dpj.*;
import java.io.*;
import COM.objectspace.jgl.*;
```
Used packages import.

```
class MyDBTree extends DBTreeClass
{public MyDBTree (Subnet net)
{super(net);}
```
Distributed binary tree is derived from library distributed binary tree implementing ParallelAlgorithm interface.

```
public void runTop() {body();}
```
This method to be invoked only at root node.

```
public void run(){
 put (recvFromParent();
 body();
}
}
```
This method is called on the nonterminal nodes. Array racived from parent node becomes a value of container element.

```
Public void body(){
 int mid_pos;
 int i,j;
 float[ ] to_sort=null;
 if (get()!=null)to_sort=(float[ ])get();
 if (to_sort!=null&&
    to_sort.length¿1){
  mid_pos=to_sort.length/2;
  float mid=to_sort[mid_pos];

  for (i=0, j=to_sort.length-1;
     i¡j;i++){
   if(to_sort[i]¿=mid)
    for (;j¿i;j–)
     if(to_sort[i]¡=mid){
     float temp =to_sort[i];
     to_sort[i]=to_sort[j];
     to_sort[j]=temp;
     if(i==mid_pos) mid_pos=j;
     break;
     }
  }
}
else mid_pos=0;

float[ ] to_left=null;
float[ ] to_right=null;
if (to_sort!=null){
 to_left=new float[mid_pos];
```
Method body partitions array being a value of container element into two parts according the *QuickSort* algorithm and sends the lower part to the left child, and the upper part - to the right child.

Detecting the middle element of the array and obtaining its value.

Array partitioning procedure: in the lower part all element values are no greater than the middle one, in the upper part - greater than middle.

In case array is empty the middle receives 0.

Definition of to_right array of apper partition and to_left array of lower partition.

```
 to_right=new float[
  to_sort.length-mid_pos];
 for (i=0; i¡mid_pos; i++)
  to_left[i]=to_sort[i];
 while (i¡to_sort.length)
  to_right[i-mid_pos]=to_sort[i++];
}
sendToChild(left, to_left);

sendToChild(right, to_right);

put (float[0]);
}
public void runBottom(){
 put (recvFromParent() );
 if (get() !=null){
  FloatArray to_sort=new
  FloatArray ((float[ ])get() );
  Sorting.sort(to_sort);
 }
}
}

class MuInput extends
DUnaryOutputIteratorClass implements
ParallelAlgorithm {
public MyInput (Dcontainer c,int[ ] m)
{super (c,m);}

public void run(){
 int number_elements;
 InputStream input=new
    FileInputStream ("numbers");
 DataInputStream fin=new;
    DataInputStream (input);
 len=fin.readInt();
 float[ ] to_sort=new float [len]
 for (int i=0; i¡len; i++)
  to_sort[i]=fin.readFloat();
 put (to_sort);
 input.close();}
 }
}
class MyOutput extends
 DUnaryForwardIteratorClass
 implements ParallelAlgorithm {
public MyOutput (Dcontainer
```

Transmitting to_left to the left child.

Transmitting to_right to the right child.

Clear element value on this node.

Leaf nodes use a standard sorting algorithm from the Java Generic Library [17].

MuInput class inherits from unary iterator and is used at the root node of distributed tree for input of array to sort from file "numbers".

Overridden run method of ParallelAlgorithm interface is called by start method of the same interface.

MyOutput class inherits from unary iterator and used at root node of distributed tree for output sorted array to the file "sorted_numbers".

```
c, int[ ] members)}
super (c, members);}
public void run(){
 OutputStream output=new
FileOutputStream (Sorted_numbers";
 DataOutputStream fout=new
 DataOutputStream (output);
 fout.writeInt(oa.lenght);
 for (int i=0; i¡oa.lenght;i++)
  fout.writeFloat (oa[i]);
 output.close();
 }
}

class PQSort {
public static void main() {

 MyDBTree tree=
 new MyDBTree(Subnet.netWorld);

 MyInput input=new MyInput (tree,
     tree.root().members() );
 input.start();
 DMultiInputIteratorClass iter =
  new DMultiInputIteratorClass(
    tree.root() );
 tree.put (ir.get() );

 do {
   tree.start(iter);




iter.advance();

 } while (!iter.atEnd() );
 MyOutput output=new MyOutput(
   tree, tree.root().members() );
 output.put (tre.gather(
     tree.root().member() ) );
```

Method main is started on all nodes of netWorld after class PQSort loading.

Definition of tree distributed over netWorld and used for sorting.

Input of the array to be sorted by means of run method of unary iterator onput of tree consisting of the root node. Definition of multiple input iterator iter of tree, having root node as a value.

Initial set up of tree elements.

Running of array partitioning procedure at the current level of distributed tree. If the level consists only of leaf nodes then standard sorting procedure is applied.

Advances distributed tree level.

Traverse all levels of the tree. Definition ofd unary iterator output of tree consisting of the root node. Method gather gathers all non-empty elements of distributed tree tree into single array at the root node, the return value of this method on other nodes is null. Resulting array is assigned as root node element value which is accessed by output iterator.

```
output.start();  | Sorted array output using method run of
}                | unary iterator output
}                |
```

To start the parallel program user should implement the followimg classes:

- Class containing the method main and representing the parallel program (e.g., PQSort from the previous example);
- The Starter class containing the method main and starting the execution of previous class.

In the method main of the Starter class the network with specified number of nodes is created, and **JavaVM**s are loaded onto these nodes. The **JavaVM**s load the first of above mentioned classes, then each **JavaVM** runs the method main of this class.

8 *The Conclusion*

The **DPJ** library provides basic capabilities of the **HP Fortran** system for the distributed data processing. The definition of networks and subnets corresponds to definition of the processor arrays in **HP Fortran**. The block distribution of the array A in **HP Fortran** corresponds to definition of the distibuted container C element of which are the blocks of the array A; the cyclic distribution of the array A in **HP Fortran** corresponds to definition of covering of the array A and distribution of it onto nodes of C, etc. In addition, the library provides more complex network structures then processor arrays in **HP Fortran**. The use of the **DPJ** library provides speedup of parallel **Java**-programs in comparison with their sequental versions. In table 2 the experiment result which uses the program implementing parallel modification of the *QuickSort* sorting algorithm are given.

Number of processors in experiment	Avarage time of execution, milliseconds	Speedup in comparison with sequential version
1	7237	1
2	3880	1,9
3	3051	2,4

As the execution speed of the program on **JavaVM** is essentially less then that of the program in machine codes, then the execution of the parallel program on **JavaVM**s concedes to that of the parallel program in a paral-

lel computer machine codes. As it was mentioned in introduction there are different ways to improve performance of the parallel program developed using **DPJ**. The more detailed discussion on the problem of the parallel **Java**-program performance improvement is out of the scope of this paper.

Acknowledgement

The project is suported by RFBR grant No 96-01-01280 and INTAS-RFBR grant No 95-0369.

References

1. J. Gosling, "The **Java** Language Environment", white paper, Sun Microsystems, Mountain View, Calif., 1995;
   http://java.sun.com
2. Ted G. Levis, *Foundation of Parallel Programing: A Machine Independent Approach,* IEEE Computer Society Press, Los Alamitos, CA, 1993.
3. "HPCC and **Java**", A Report by The Parallel Compiler Runtime Consorium (PCRC), incomplete draft, May 12 1996;
   http://www.npac.syr edu/users/gcf/hpjava3.html
4. HPF:High Performance Fortran Language Specification, High Performance Fortran Forum, version 2.0, January 31 1997;
   http://www.crpc.fice.edu/HPFF/hpf2/index.html
5. E. Johnson, P. Beckman, D. Gsnnon, *"HPC++: An Experiment with the Parallel Standard Template Library"*,
   http://www.cs.indiana.edu/hyplan/ejohnson/papers/pstl.html
6. F. Bodin, P. Beckman, D. Gannon, S. Narayana, Sh. X. Yang, "Distributed pCC++: Basic Ideas for an Object Parallel Language", *Scientific Programming*, Vol. 2, No3, 1993;
   http://www.extreme.indiana.edu/sage/docs.html
7. MPC++ Version 2: Massively Parallel, Message Passing, Meta-level Processing C++
   http://www.rwcp.or.jp/lab/mpslab/mpc++/mpc++.html
8. Susan F. Hummel, Ton Ngo, Harini Srinivasan, "SPMD" Programming in **Java**", tech. report, IBM T.J. Watson Research Center;
   http;//www.npac.syr.edu/projects/javafocse/cpande/IBMspmdjava_new.ps
9. Marc A. Hamilton, "**Java** and the Shoft to Net-Centric Computing", *Computer*, IEEE Computer Society, August 1996, vol. 29, No 8, p.31-39.
10. MPI: Message Passing Interface Standard, Message Passing Interface Forum, June 12 1995;
    http://www.mcs.anl.gov/mpi/index.html

11. Frank Yellin, "The Java Mative Code API", Sun Microsystems, Mountain View, Calif., 1995;
http://www.javasoft.com/doc/jit_interface.html

12. *Microsoft SDK for Java 1.5*, Microsoft Corporation, Palo Alto, CA;
http://www.microsoft.com/java/sdk

13. Aart J.C. Bik, Dennis B. Gannon, "Automatically Exploiting Implicit Parallelism in Java", tech. report, Computer Science Dept., Indiana University, Indiana;
http://www.extreme.indiana.edu/ ajcbik/JAVAR/index.html

14. V. Ivannikov, S. Gaissaryan, M. Domrachev, V. Etch, N. Shtaltovnaya, "DPC++:C++ Class Library for Development of Data-Parallel Programs" in "*Voprosy Kibernetiki. Prilojeniya Sistemnogo Programmirovanija*", Moscow, NSK RAS,1997, (in russian)

15. D. R. Musser, A. Saini, "*TSL Tutorial and Reference Guide. C++ Programming with the Stendard Template Library*", Addison-Wesley, 1996;
http://www.aw.com/cp/musser-saini

16. Draft Proposed International Standard for Information Systems - Programming Language C++, Accredited Standards Committee* Doc No: X3J16/96-0225 X3,INFORMATION PROCESSING SYSTEMS WG21/N1043 Date: 2 December 1996 Project: Programming Language C++,
http://www.warwick.ac.uk/c++/pub/wp/html/cd2/index.html

17. R. Rew, "The Java Genric Library (JGL)", Boulder Java Users Group, UCAR Unidata, September 1996,
http://www.unidata.ucar.edu./staff/russ/java/jgl-bjug