Parallel Processing in heterogeneous cluster architectures using JavaPorts

by Demetris G. Galatopoullos
and Elias S. Manolakos

Introduction

The variety of individual machine architectures, the high availability of spare CPU time and the recent advances in high-speed networks technology have made *clusters* of workstations a promising and cost effective alternative to conventional parallel machines, primarily for coarse grain parallel algorithms where the communication among processors is not excessive. The objective of the JavaPorts project is to provide a user friendly environment so that programmers who are not intimately familiar with *object oriented* and *concurrent programming* concepts, can still develop, map and easily reconfigure parallel and distributed applications written in Java [1].

The JavaPorts environment relieves the programmer from the details of processors coordination while saving the large effort of developing the application entirely from scratch every time the cluster topology changes. The developer specifies a *Task Graph* with the aid of a simple configuration language or a graphical user interface. In this graph, *Tasks* are allocated to compute *Nodes* (machines) and may communicate via *Ports* . A *Port* is a software abstraction that appears as a mailbox to a task that may write to (or read from) it in order to send (receive) a message. Its internal operation, which is not a responsibility of the programmer, guarantees the safe and reliable point-to-point communication among two connected tasks. Writes to (as well as reads from) a port are *anonymous* , meaning that the name of the task that should receive (has sent) the message does not need to be known.

When a Task Graph is specified, the system will automatically create Java program *templates* , one for each defined task, that the developer can use to complete the coding of the specific algorithm. In these templates the part of the code required to create and register defined ports is automatically generated. If at some later point in time the user decides to modify the allocation of tasks to nodes, or to add new ports to a task, the JavaPorts tool can be called to process again the Task Graph and update all affected templates, without touching the user-added part of the code. In this fashion, software components for concurrent Java applications may be built incrementally and reused in a very simple and natural manner.

As the first part of its name suggests, the JavaPorts system is based on Java [1] ,an object-oriented, easy to learn programming language that has become very popular. Java compilers produce an intermediate format, the bytecodes, which is interpreted and may be executed on any type of architecture and running a Java Virtual Machine (JVM) [2]. JVMs are abstract computing machines that may run on top of any existing operating system. Among the many interesting features of the Java language, the one most directly related to concurrent computing is its built-in *threading* capability [3]. The user-defined tasks are executed as threads (lightweight processes) allocated to the specified nodes of the cluster and used to initiate independent activities that may communicate by exchanging user-defined messages. The second part of the name denotes the added capability of software Ports, entities which provide the services of a communication interface to the tasks. Ports are introduced to achieve reliable transport of user-defined messages among tasks while hiding from the programmer all the details of their internal operation. JavaPort classes implement an interface which provides to the user the necessary methods for accessing port services, such as reading and writing messages.

The JavaPorts model

The JavaPorts tasks follow the *Ideal Worker Ideal Manager* (IWIM) model of *anonymous communications* i.e. a task does not need to know the identity of the tasks with which it exchanges information, in contrast with the Targeted-Send/ Receive (TSR) model [4,5]. Every task may be thought of as an Ideal Worker who does not care where its input comes from, nor where its output goes to; the worker simply does the job provided that the expected inputs are received at the designated ports when they are needed. So tasks can be reused as long as their ports structure remains the same. The synchronization among the ports and the creation and management of their associated threads is handled automatically by the JavaPorts system classes and not by the programmer.

However, the definition of the point-to-point connectivity which allows the *blind* passing of messages among computing entities is user-controlled. The programmer is assumed to be aware of the nodes (machines) in the cluster and performs the static allocation of tasks to nodes. Once this assignment is completed `JavaPorts` is responsible for returning to the programmer a Java code template for each defined task as well as for creating and initializing the necessary port objects. The user may have to add code in a template-based program in order to complete the tasks implementation for a specific parallel algorithm.

The Task Graph

The task graph provides a mechanism which allows the developer to map the work partitions of the application onto the distributed nodes of the cluster. A sample Task Graph is shown in Figure 1, where boxes represent tasks and dashed boxes nodes (machines) respectively. There are three user-defined tasks, two allocated on Node1 (Task1 and Task2) and one on Node2 (Task3). The two tasks that are local to Node1 utilize a pair of ports to communicate and the same is true for Task1 and Task3 that are remote to each other.

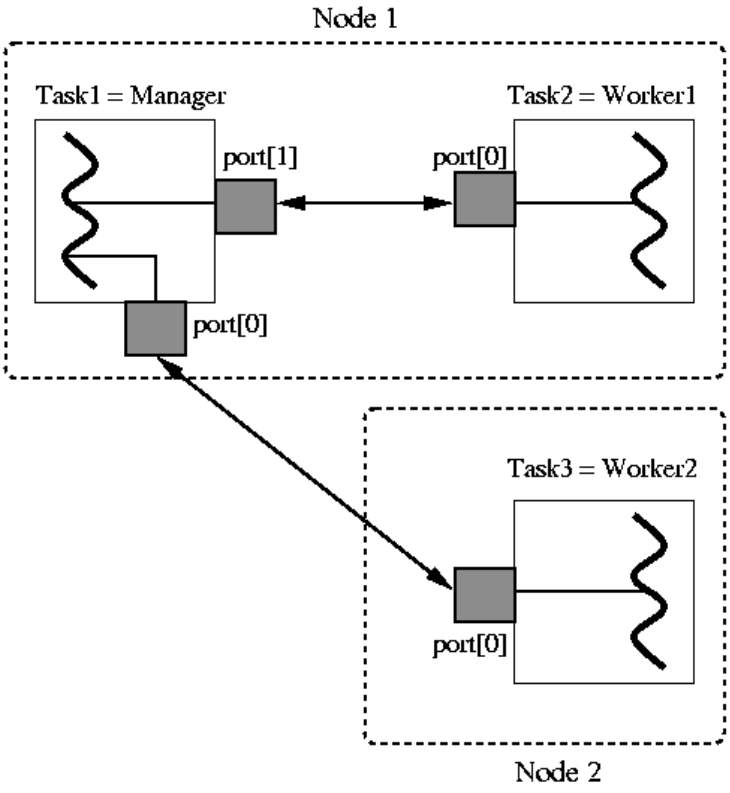

Figure 1. A sample Task Graph.

JavaPorts will read the corresponding user-defined configuration file and return the chosen parameters to the system's classes that will be used to generate or update the task templates. The configuration file corresponding to the above Task Graph is shown in Figure 2. The user may alter the allocation of tasks to nodes, without having to re-write any code she added in an existing template, and have the parallel algorithm executed on a different cluster setup. JavaPorts will implement all the template updates needed to run the concurrent tasks in the new setup, even if all of them are assigned to the same node.

```
begin configuration
  begin definitions
    define     machine Node1= "corea.cdsp.neu.edu"
    define     machine Node2= "walker.cdsp.neu.edu"
    define     task  Task1= "Manager"
    define     task  Task2= "Worker1"
    define     task  Task3= "Worker2"
  end definitions
  begin  allocations
    allocate   Task1 Node1
    allocate   Task2 Node1
    allocate   Task3 Node2
  end allocations
  begin connections
    connect   Task1.port[0] Task3.port[0]
    connect   Task1.port[1] Task2.port[0]
  end connections
end configuration
```

Figure 2. Configuration file for the Task Graph of Figure 1.

User Program Templates

The templates generated for Task1 (called Manager) and Task3 (called Worker2) are detailed in Figure 3 and Figure 4 respectively. The parts in bold letters are generated by JavaPorts and what comes in between them corresponds to user-added code. It is assumed that the Manager task forms a data message that is sent to both worker tasks and then continues to do some local computation. Each worker task waits until it receives data and then uses the data to produce some results that are sent back. Then the manager collects the results returned by the workers.

```
// Code parts in bold are generated by the JavaPorts system

public class Manager
run()
{
  Init InitObject = new Init();
```

```
  InitObject.configure(port);

  // User-added algorithm implementation starts here

  // Form the data  message and send it out
  Message msg1 = new Message(UserData);
  port[0].AsyncWrite(msg1,key1);
  port[1].AsyncWrite(msg1,key2);

  // Code for any local processing may go here

  // Get References to expected results
  ResultHandle1 = port[0].AsyncRead(key3);
  ResultHandle2 = port[1].AsyncRead(key4);

  // Code to retrieve results may go here
}


public static void main()
{
  Runnable ManagerThread = new Manager();
  Thread manager = new Thread(ManagerThread);
  manager.start();
}
```

Figure 3: Java code template for the Manager task.

```
public class Worker2
run()
{
  Init InitObject = new Init();
  InitObject.configure(port);


  // Get a Reference to the object where the incoming data message will be stored
  ListHandle = port[0].AsyncRead(key1);

  // Use it to wait, if needed, until the data has arrived
  DataHandle = ListHandle.PortDataReady();

  // The algorithm that uses the incoming data and produces results goes here.

  // Prepare a message with the results
```

```
  Message msg1 = new Message(ResultData);

  // Write back the results
  port[0].AsyncWrite(msg1,key3);

}

public static void main()
{
  Runnable Worker2Thread = new Worker2();
  Thread worker = new Thread(Worker2Thread);
  worker.start();
}
```

Figure 4: Java code template for the Worker2 task.

The ports creation is handled by a call to the`configure()` method of the`Init` object. The`Init` class is part of the`JavaPorts` system and its implementation is hidden from the user. This object is also responsible for making transparent remote port lookups and the local port object registrations implemented using the`Remote Method Invocation` (RMI) [6] package. Since these are expensive operations, the`JavaPorts` system manages them at the task startup time. Therefore, the programmer's code execution is freed from any delays while computations and distributed communications are being performed.

The templates, are generic Java classes that represent runnable objects. The user is provided with a complete`main()` method which creates and initiates a task thread to run the task object, as depicted at the bottom bold-face part in Figure 3. The user's responsibility is to fill in the rest of the`run()` method of a task template in order to implement the application specific part of the task. If a task needs to communicate the user can use calls to publically known read and write methods (that form a Java interface) on the corresponding port objects. Reading from and writing to a port is asynchronous (non-blocking) and is implemented by`JavaPorts` classes.

A task may create several subtasks which will carry out the required computations, depending on the available concurrency and the user's willingness to exploit it. Moreover, multiple`main()` tasks on a node, may still communicate through ports (see Figure 1). The user simply accesses the local ports of a task for reads and writes. The`JavaPorts` system has the ability to identify when the`RMI` runtime environment must be called for passing messages to a remote port object, without the programmer having to be aware of it. So the same code templates may be used for concurrent computing on the one node (machine) or on a number of distributed nodes in the cluster. In the former case the concurrent tasks are time sharing the same CPU and in the latter they may run truly in parallel. So additional hardware resources, if they become available, may be fully exploited for improving the performance, without having to incur any additional development costs for re-engineering the user code of the application.

The templates provided to the user are automatically defined as part of the`JavaPorts` class package. Therefore, after compilation they are ready to execute along with the system classes. In the cluster of workstations which run Network File Systems it is not needed for the programmer to distribute the various task executables since they are visible by all the nodes in the cluster.

The Ports interface

The Java programming language introduced the concept of an interface in order to deal with multiple inheritance. A class will implement all the methods included in an interface. In `JavaPorts` an interface is used to provide the user with a set of allowable operations which may be applied on the ports for the purpose of exchanging messages, while keeping their implementation hidden. This programming method provides a security feature ensuring the proper operation of the system. The Java port interface currently includes the following methods that can be called on port objects.

- `AsyncWrite(Msg, MsgKey)` : It will write the `Msg` message object to a port along with a key for personalizing the message. This method will execute concurrently with the task that calls it (non-blocking write).
- `AsyncRead(MsgKey)` : It signifies the willingness of the calling task to access a certain message object expected at this port. By executing this method, the task will obtain a reference to an object in a list data structure maintained by the port where the message will be placed upon arrival. This reference can be used to check if the message has been received.
- `PortDataReady()` : It is an accessor method to the objects of the port list. When this method is called and if the message has arrived at the port, it returns a reference to the object where the incoming data resides; otherwise it waits until the message arrives.

The messages which are communicated among tasks are instances of the user defined class `Message` . This class extends the `Object` class giving the user the flexibility to include any desired type of information in the message. In combination with the user-defined templates, this has critical reusability implications in the `JavaPorts` design. The ensuring of correct delivery of messages relies on a user-defined variable, the `MsgKey` . Caution must be used by the programmer in order to use unique matching keys for the same data traveling among tasks.

Anonymous message passing using `JavaPorts`

The internal operation of the `JavaPorts` system during a communications session between two tasks residing on different nodes is described below.

Both the Manager and the Worker task threads begin their operation by initializing their ports and by pairing them up. This is the responsibility of the `Init` object in each task. The method `configure` will create, register and lookup a port with RMI and return a reference to it back to the calling thread. In reality, this method will return an array of ports, each one of them configured and registered. Upon creation, each port is associated with a linked list of objects that serves as a depository of incoming messages from the port connected to it.

Due to the symmetry of the operation, only the communication from the Manager (sending) to the Worker (receiving) task thread is described here. The Manager task thread issues an `AsyncWrite` to its associated port with a user-defined key. This key will identify the data on the remote side. The `AsyncWrite` method spawns a secondary thread, the `PortThread` , and returns. This thread will be responsible for transporting the data to the remote port asynchronously and without blocking the Manager task thread. The `PortThread` calls the `receive` method of the remote port object. This method makes use of the RMI serialization mechanism in order to transfer the message to the remote node. The incoming data will be inserted in the appropriate object element of the remote port list.

On the receiving node, the Worker task thread issues an `AsyncRead` to its associated port using the corresponding key of the expected incoming data. The method will return a reference to the specific object in the list where the incoming

message will be deposited upon arrival. At the point in time when the Worker thread needs this data, it calls the method`PortDataReady` on the specified list object and then enters the wait state. The`receive` method, which the Manager`PortThread` calls during the transportation of data, inserts the incoming data to the corresponding list object and then notifies all the waiting threads in the address space of the receiving node. Once the Worker thread is notified, it exits the wait state and checks whether the incoming data has the correct key. If not it re-enters the wait state, otherwise it retrieves the data. The exact same protocol may be used in the reverse direction in order to transfer a message the Worker to the Manager thread.

Related Work

There are several on-going research projects around the world aiming at exploiting or extending the services of Java to provide frameworks that may facilitate parallel applications development. Among them we mention here the `Java//` [7, 8] and `Do!` [9] projects that employ *reification* and *reflection* to support transparent remote objects and high level synchronization mechanisms. The`JavaParty` [10] is also a package that supports transparent object communication by employing a run time manager responsible for contacting local managers and distributing objects declared as`remote` . (JavaParty minimally extends Java in this respect). The`Javelin` [11] project tries to expand the limits of clusters beyond local subnets, thus targeting large scale heterogeneous parallel computing. It is motivated from the idea of utilizing available CPU resources around the Internet for solving compute-intensive tasks.`Javelin` is composed of three major parts: the broker, the client and the host. Processes on any node may assume the role of a client or a host. A client may request resources and a host is the process which may have and be willing to offer them to the client. Both parties register their intentions with a broker, who takes care of any negotiations and performs the assignment of tasks. Message passing is a major bottleneck in`Javelin` because the messages destined for remote nodes traverse the slow TCP or UDP stacks utilized by the Internet. However, for compute-intensive application s, such as parallel raytracing, `Javelin` has demonstrated good performance.

The main conceptual difference between our work and the above mentioned projects is that`JavaPorts` impose on the programmer the view of a parallel application as a collection of interconnected concurrent tasks with clearly marked boundaries and a well structured interface mechanism. This view is inspired by how complex digital systems are designed today using hardware description languages, such as`VHDL` [12]. A simple configuration language provides the means of describing a work distribution solution to an application, while giving the user the control over "what-goes-where". When a configuration is decided the`JavaPorts` system produces reusable software components, the task templates. By using ports, these components may establish bi-directional asynchronous and anonymous point-to-point communication. The messaging scheme is user-defined and general since the system is able to handle all possible types of information flow. The developer may generate as many components as needed for an application through generic class templates.`JavaPorts` attempt to provide a simple and user friendly environment for the rapid prototyping of parallel and distributed applications built by well-structured high quality components, in which hiding the coordination and communication details does not come at the expense of severe performance degradation.

Experimental Results

In this section we simply outline some experiments conducted using the`JavaPorts` system and we present only some indicative performance results (A more detailed evaluation is outside the scope of this paper and will appear later). As an illustrative example, we have chosen the matrix-vector multiplication problem applied to a variety of two-node configurations, as it was also done by others (see [7] and [13]). The multiplicand is a square matrix whereas the multiplier is a single column vector. We conducted the multiplication using two concurrent tasks (a manager and a worker) allocated to two different nodes. The part of the multiplicand matrix needed by the worker task resided on the

remote node before the commencement of the experiment. The communication overhead, i.e. the time it takes for the multiplier vector to be sent from the manager to the remote worker task, as well as the time it takes for the partial product vector to be sent back from the worker to the manager was measured and taken into consideration in our timing analysis.

In the conducted experiment, a 1000-by-1000 matrix is multiplied by a 1000-by-1 vector. The`CurrentTimeMillis` method of the`System` class (which returns the current time in milliseconds) was used to gather measurements. We acquired the time at the beginning and at the end of each targeted period and the elapsed time was determined by calculating the difference of the two measurements. The`JavaPorts` system was configured to run on a set of two Sun Sparc-4 Microsystems workstations which belong to a local standard Ethernet subnet running Solaris version 2.5.1.

In the two-node scenario, the number of rows of the multiplicand matrix which resided on each of the two nodes was varied from 0 to 1000. The manager task starts first, sends the multiplier vector to the remote worker, then performs its local computation and finally collects back the partial result returned by the worker and assembles the overall product vector. In Figure 5, the solid line depicts the total time needed by the manager task to complete as a function of the number of rows allocated to the remote worker task. The dashed lines show the times needed for the computation of the local partial product by the manager and the time the manager should wait subsequently for the remote partial product vector to arrive from the remote worker task. In essence, the decomposition of the total manager time to its main sequential components verified that as the work allocated to the remote worker increases, the manager local computation time decreases and the waiting time (for worker results) increases, as expected. achieved close to the point where the rows of the matrix are split evenly among the two compute nodes. This minimum time corresponds to 62% of the time needed to solve the problem (1000 rows) using a single thread Java program in one machine i.e. the achieved speedup using two concurrent tasks in two nodes is about 1.6, which is very good considering that the network is a standard 10Mb shared Ethernet and the problem (message) size not very large.

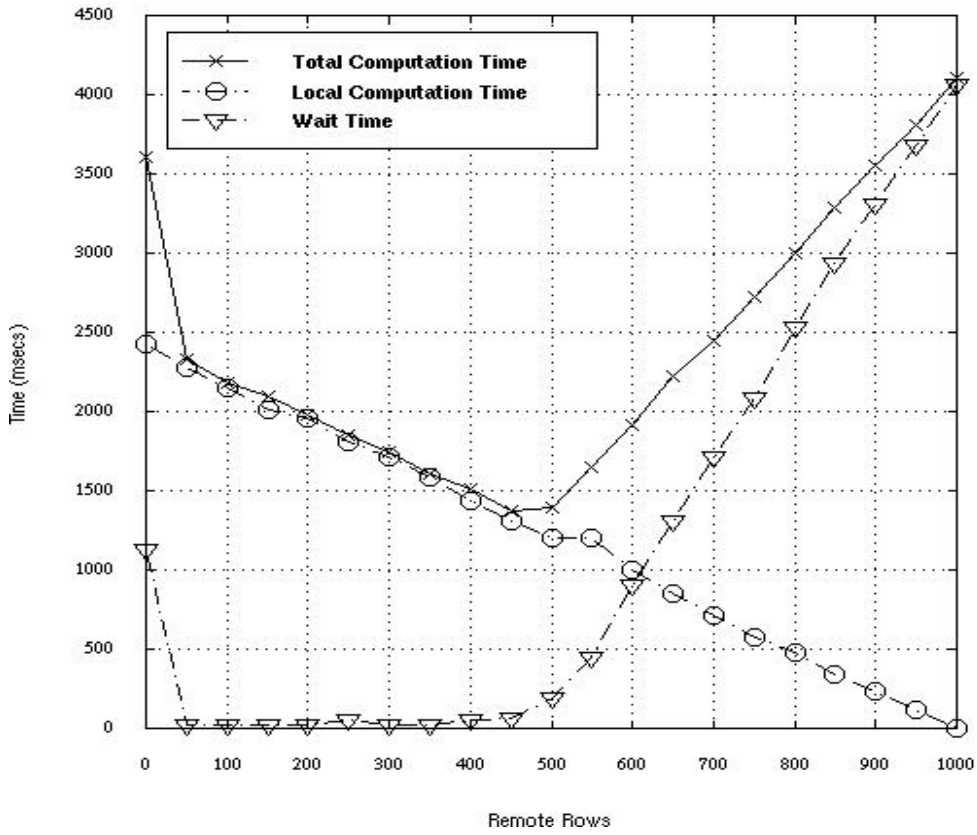

Figure 5. Time of the manager task vs. rows allocated in the remote worker task, in a two-node schenario.

We have also conducted the same experiment on a three-node cluster. In this experiment, the multiplicand matrix was split among three nodes. The manager node is assigned a number of local rows which varies from 0 to 1000. The remaining rows are split in various ways between two worker nodes. In this scenario, the machines which run worker tasks were chosen to possess identical hardware characteristics. Similar to the two-node scenario, there is a minimum observed close to the point where the remote tasks collectively have as many matrix rows to multiply with the vector as the manager task. Using three nodes leads to speadup, relatively to using two nodes, when 700 or more elements of the product are calculated by the remote workers.

For the two-node solution, the`JavaPorts` performance was also compared to that of two other major designs, the`Java//` [6] and the`JavaParty` [9]. Similar to our system, the`JavaParty` application was ran on two Sparc-4 nodes whereas the`Java//` experiment [6] made use of two UltraSparc stations. To be able to obtain the`JavaParty` times, a single thread Java program was written. Following the guidelines of the`JavaParty` group, the worker task was declared as`remote` , thus allowing the system to migrate the worker task object to the remote node. As it can be seen from the plot in Figure 6, the least time achieved occurs at approximately the same number of remote rows for all designs. Among all designs,`JavaPorts` exhibited the best performance across the full range of rows allocated to the remote worker node.

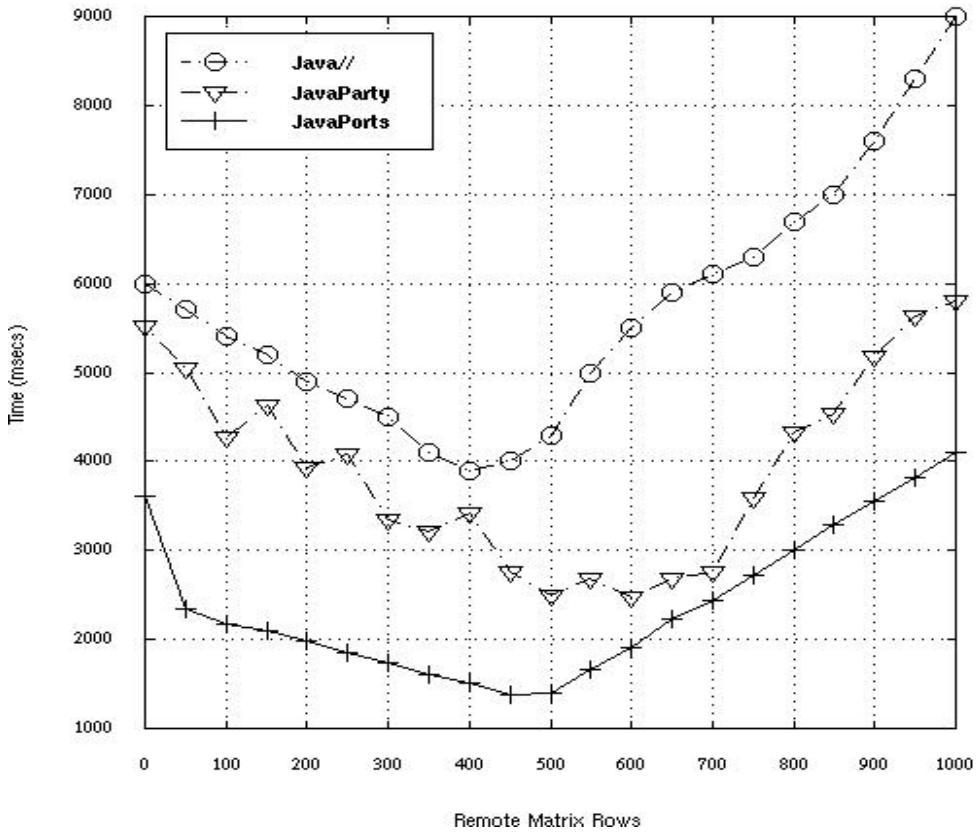

Figure 6. Comparison of three designs using the same matrix-vector multiplication problem.

Conclusions

We have introduced`JavaPorts` , an environment for flexible and modular concurrent programming on cluster

architectures. The design encourages reusability by enabling the developer to build parallel application using Java in easy modular steps. The components which comprise such designs may be manipulated through the system and without modifying any existing user code they may be be used to reconfigure and run a parallel application in a completely new cluster setup. The user-defined message object may be altered to customize the needs of transferring any type of data across the network. By giving the developer such capabilities, the `JavaPorts` environment may be customized for specific client-server applications where a large variety of services may be available. We managed to keep the task of tracking remote objects and marshalling of data between different address spaces transparent to the developer. The user is not aware of how the data is communicated among nodes in the cluster.

Currently there is a considerable amount of on-going work. The completion of the Graphical User Interface for allowing the developer to enter and modify a task graph in `JavaPorts` is our first priority. This tool will provide the developer with a visual sense of how the application is configured to run on a cluster. This is an important issue in parallel computing where data partition among nodes is fairly easier to understand when it is depicted visually. In addition we are improving the configuration language that provides a text based approach to the allocation of tasks to nodes. Experienced users may find this representation convenient for the development and customization of their application.

We are also designing large scale applications using a variety of common logical node configuration patterns (star, pipeline, mesh). In addition to compute-intensive applications we consider other domains, such as distributed network management, distributed simulation and embedded systems, that may take advantage of the easy application configuration capabilities of `JavaPorts` . Some of these applications are selected to allow us to fully test and optimize specific aspects of the design, before it is released for general use.

Acknowledgements

The authors would like to thank Prof. Bernd Freisleben of Siegen University, Germany who was on sabbatical leave at the *Parallel Processing and Architectures* group of Northeastern University and provided valuable input during the early stages of the project.

References

1

Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley Longman, Inc., 1996.

2

The Java Virtual Machine Specification. http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html

3

S. Oaks and H. Wang. *Java Threads.* O'Reilly Publ., 1997.

4

F. Arbab. The IWIM model for coordination of concurrent activities. *Coordination '96, Lecture Notes on Computer Science,* vol.1061, April 1996.

5

F. Arbab, C.L. Blom, F.J.Burger and C.T.H.Everaas. Reusability of Coordination Programs. Technical Report CS-R9621, Centrum voor Wiskunde en Informatica, The Netherlands, 1996.http://www.cwi.nl/cwi/publications/reports/abs/CS-R9621.html

6

Remote Method Invocation Specification. Sun Microsystems, Inc., 1996/1997. http:////www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html

7

Denis Caromel and Julien Vayssiere. A Java Framework for Seamless Sequential, Multi-threaded, and Distributed Programming. *ACM Workshop: Java for High-Performance Network Computing*, pages 141-150, February 1998.

8

Proactive PDC - Java//. http://www.inria.fr/sloop/javall/index.html .

9

Pascale Launay and Jeab-Louis Pazat. Generation of distributed parallel Java programs. Technical Report 1171, IRISA, France, 1998. http://www.irisa.fr/EXTERNE/bibli/pi/1171/1171.html

10

Michael Philippsen and Matthias Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225-1242, November 1997.

11

Bernd O. Christiansen et al. Javelin: Internet Based Parallel Computing Using Java. *Concurrency: Practice and Experience*, 9(11):1139-1160, November 1997.

12

Z. Navabi. *VHDL: Analysis and Modeling of Digital Systems.* McGraw Hill, 1998.

13

Rajeev R. Raje, Joseph I. William, and Michael Boyles. An asynchronous Remote Method Invocation (ARMI) mechanism for Java. *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997. http://www.npac.syr.edu/projects/javaforcse/acmprog/25.ps

---

Demetris Galatopoullos is a PhD student of Electrical and Computer Engineering at Northeastern University. He received his BS and MS degrees from Boston University in the area of Computer Engineering. He is a member of the Parallel Processing and Architectures Research Group at Northeastern University. His technical and research areas of interest include distributed and parallel object-oriented design, software component programming, large-scale software design and process management, and artificial intelligence.

Elias S. Manolakos received the PhD degree in Computer Engineering from University of Southern California in 1989, the MSEE degree from University of Michigan, Ann Arbor, and the Diploma in Electrical Engineering from the National Technical University of Athens , Greece. As a student he received the first Pan-Hellenic Award in Computer Science and Engineering of the Greek State Scholarships Foundation (IKY) in 1984, and as Assistant Professor an NSF Research Initiation Award in 1993. Prof. Manolakos is currently an Associate Professor at the Electrical and Computer Engineering Dept. of Northeastern University, where he is leading the Parallel Processing and Architectures Research

Group . His current research interests include: High Performance Computing, Synthesis of Parallel Algorithms and Architectures, Pattern Recognition and Bioinformatics. His research has been supported by NSF, DARPA and DOE. Dr. Manolakos is a Senior Member of IEEE and has been the program chair in several conferences. He is serving in the editorial boards of the *IEEE Computational Science and Engineering Magazine*, the *IEEE Transactions on Signal Processing* etc. He has authored, or co-authored with his students, more than 50 refereed publications and co-edited three books.